计算机图形学
原理、算法及实践

李晓武　主编

清华大学出版社

计算机图形学

——原理、算法及实践

主　编　李晓武
副主编　樊百琳　曹　彤

清华大学出版社
北　京

内容简介

计算机图形学是计算机科学的一个重要分支，具有较强的理论性和实践性。本书内容丰富，不仅系统介绍了计算机图形学的主要研究内容以及基本原理，也提供了大量的编程实践，在一定程度上帮助读者开发真实的图形应用程序。理论与实践相结合是本书的重要特色。该书不仅系统讲解了真实图形开发环境下的 OpenGL 技术，也提供了 Web 环境下的图形开发方法，可以使读者了解计算机图形学的应用趋势。

本书编程所用数据均通过动态交互获取，而非提前设定，因此，最后的图形显示效果为实时的结果，这样，也直接验证了书中算法的稳定性、可靠性和可行性。

本书的读者对象可以是在校本科生、研究生，也可以是希望学习和掌握计算机图形学的相关人员。本书可以作为计算机图形学的教材，也可以作为学习计算机编程的技术书籍。

图书在版编目（CIP）数据

计算机图形学：原理、算法及实践/李晓武主编. —北京：清华大学出版社，2018
ISBN 978-7-302-49873-5

Ⅰ. ①计… Ⅱ. ①李… Ⅲ. ①计算机图形学 Ⅳ. ①TP391.411

中国版本图书馆 CIP 数据核字(2018)第 052454 号

责任编辑：许 龙 赵从棉
封面设计：常雪影
责任校对：赵丽敏
责任印制：丛怀宇

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 **邮 编**：100084
社 总 机：010-62770175 **邮 购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经 销：全国新华书店
开 本：185mm×260mm **印 张**：25.75 **字 数**：626 千字
版 次：2018 年 6 月第 1 版 **印 次**：2018 年 6 月第1 次印刷
印 数：1～1500
定 价：65.00 元

产品编号：078796-01

前言

FOREWORD

计算机图形学是计算机科学的一个重要分支，现在的计算机技术应用领域，例如计算机辅助设计(CAD)、计算机动画仿真、影视广告特效制作、电脑网络游戏、虚拟现实、三维扫描和打印以及人工智能技术等涉及图形图像的方面，都在直接或者间接地使用计算机图形学的理论和方法。那么，如何学习才能掌握计算机图形学这门学科呢？

作为一门计算机应用技术，计算机图形学具有较强的理论性和实践性。本书内容丰富，不仅介绍了计算机图形学这门学科的主要研究内容以及基本原理，也提供了大量的编程实践，理论与实践相结合是本书的重要特色，可以在一定程度上帮助读者开发真实的图形应用程序；该书不仅系统讲解了真实图形开发环境下使用的 OpenGL 技术，也提供了 Web 环境下的图形开发方法，可以使读者了解计算机图形学的应用趋势。

在结构安排上，本书分为两大部分。

第一部分为本书前 9 章内容，详细介绍了计算机图形学的主要研究内容、基本原理以及图形学的开发环境和编程实践。具体如下：

第 1 章　绪论

介绍了计算机图形学的基本概念、相关研究内容、应用领域、发展历史以及发展趋势。

第 2 章　图形开发工具及使用

详细介绍了图形开发工具 VC++系统的开发流程、相关函数及集合的使用、基本像素点的绘制方法以及非模式对话框的交互式实现方法。

第 3 章　基本图形的生成

详细讲解了直线扫描转换生成的各种算法及实现、VC++的橡皮筋和双缓存交互技术、圆及圆弧的扫描转换、椭圆的扫描转换、多边形的扫描转换和填充、字符的表示、线宽和线型处理方法以及反走样技术。

第 4 章　裁剪

详细讲解了直线在矩形窗口的裁剪算法、多边形的裁剪(包括矩形及凸多边形裁剪窗口的裁剪和任意形状多边形的裁剪及实现)、圆的裁剪(包括圆形窗口的直线裁剪和任何多边形窗口对圆的裁剪及实现)以及字符裁剪。

第 5 章　图形变换

介绍了图形变换的数学基础，详细讲解了二维图形的矩阵变换、交互式对象拾取和捕捉技术、三维图形的矩阵变换、三维图形的线框拉伸造型方法、投影变换、三维图形的交互拾取以及透视投影变换。

第 6 章　消隐技术

介绍了消隐的相关概念，详细讲解了凸多面体的可见性判断方法和一般多面体的各种消隐算法，并对基于扫描线的消隐算法进行了详细分析和编程实现。

第 7 章　真实感图形绘制

介绍了颜色与光的基本知识，详细讲解了简单光照模型和复杂光照模型，并实现了简单光照模型的编程，讲解了纹理映射的实现原理。

第 8 章　曲线曲面

介绍了曲线曲面的参数表示方法以及相关概念，详细讲解了 Bézier 表示法、递推生成算法和 Bézier 曲面，详细讲解了 B 样条的定义、B 样条曲线的递推定义，介绍了 B 样条曲线的绘制以及 B 样条曲面的拉伸造型方法，介绍了非均匀有理 B 样条 NURBS 方法。

第 9 章　计算机动画与仿真

介绍了动画的概念及基本原理、逐帧动画和实时动画实现方法，并对实时动画方法进行了编程实现。

这部分内容各章节之间理论独立，但是代码编程部分前后关联，并创建了一个集成的图形程序，因此，在练习这一部分的代码时，建议循序渐进，不要跳跃式学习。

本书第二部分为第 10 章和第 11 章，这两章各自独立，分别介绍了真实环境下计算机图形学应用程序接口（API）——OpenGL 技术和 Web 环境下的图形开发技术，如果读者仅对 OpenGL 技术或者 Web 图形开发感兴趣，可以直接跳过本书前面的章节，学习这两章的内容。具体如下：

第 10 章　基于 OpenGL 的图形开发技术

详细讲解了 VC++环境下 OpenGL 的配置方法、OpenGL 基本图形及真实感图形绘制技术、OpenGL 图像处理技术、OpenGL 纹理映射技术以及 OpenGL 曲线曲面技术等。

第 11 章　Web 图形开发技术

介绍了 Web 绘图技术所需的 Html 文档结构、JavaScript 脚本语言和 canvas 功能标签，详细介绍了 JavaScript 语言生成基本图形的编程方法和基于 WebGL 的 3D 图形技术。

本书编程所用数据均由应用程序通过动态交互获取，而非提前设定，因此，最后的图形显示效果为实时的结果，这样，直接验证了书中算法的稳定性、可靠性和可行性。

本书的读者对象可以是在校本科生、研究生，也可以是希望学习和掌握计算机图形学的相关人员。本书可以作为计算机图形学的教材，也可以作为学习计算机编程的技术书籍。

本书第 1～10 章主要由李晓武编写，第 11 章主要由周晓雨编写。由李晓武担任主编，樊百琳和曹彤担任副主编，参编人员还有万静、杨皓、陈平、许倩、陈华和杨光辉等，他们还提出了非常宝贵的意见，在此一并致谢。除了已经列出的参考文献以外，还参考了其他相关图书和网上资料，无法一一列出，谨向所有作者表达谢意。本书的编写和出版得到了北京科技大学教材建设经费的资助，在此也表示感谢。

本书中编程实现的应用程序代码及其他相关资料可通过本书封底的二维码扫描下载。

由于作者水平有限，错漏之处在所难免，恳请读者批评指正。

编　者

2018 年 4 月于北京

目录

CONTENTS

第一部分

第1章　绪论 …… 3

1.1　概念 …… 3
1.2　研究内容及应用领域 …… 3
　1.2.1　图形和图像的关系 …… 3
　1.2.2　图形输入输出硬件技术 …… 4
　1.2.3　计算机图形学的主要研究内容 …… 4
　1.2.4　计算机图形学的应用领域 …… 6
1.3　发展历史 …… 8
　1.3.1　萌芽阶段 …… 8
　1.3.2　发展阶段 …… 8
　1.3.3　推广应用阶段 …… 9
　1.3.4　实用化阶段 …… 9
　1.3.5　标准化、智能化阶段 …… 10
　1.3.6　多学科融合发展阶段 …… 10
1.4　学科发展 …… 10

第2章　图形开发工具及使用 …… 13

2.1　VC++开发系统简介 …… 13
　2.1.1　VC6.0系统介绍 …… 13
　2.1.2　VC++相关设计流程 …… 18
2.2　VC++基本图素的绘制方法 …… 21
　2.2.1　相关类及函数 …… 21
　2.2.2　基本像素点的交互式绘制方法 …… 22
　2.2.3　非模式对话框交互式实现方法及颜色对话框的使用 …… 23

第3章　基本图形的生成 …… 29

3.1　直线的扫描转换 …… 29
　3.1.1　直线扫描转换原理 …… 29
　3.1.2　数值微分法 …… 30
　3.1.3　中点画线算法 …… 33
　3.1.4　Bresenham画线算法 …… 36
　3.1.5　图形程序设计及VC++的橡皮筋和双缓存交互技术 …… 39

3.2 圆的扫描转换…… 44
3.2.1 圆的扫描转换概述 …… 44
3.2.2 中点画圆算法 …… 45
3.2.3 Bresenham 画圆算法 …… 46
3.2.4 圆弧段的扫描转换 …… 48
3.3 椭圆的扫描转换…… 54
3.4 多边形的扫描转换及区域填充…… 57
3.4.1 多边形的扫描转换 …… 57
3.4.2 区域填充 …… 69
3.5 字符和汉字的表示…… 72
3.6 线宽和线型处理…… 73
3.6.1 线宽处理 …… 73
3.6.2 线型处理 …… 76
3.7 反走样技术…… 76

第 4 章 裁剪 …… 79

4.1 点和直线的裁剪…… 79
4.1.1 点的裁剪 …… 79
4.1.2 直线裁剪 …… 79
4.2 多边形裁剪…… 85
4.2.1 多边形裁剪概述 …… 85
4.2.2 矩形及凸多边形裁剪窗口裁剪 …… 85
4.2.3 任意形状多边形的裁剪 …… 95
4.3 圆裁剪 …… 107
4.3.1 圆裁剪概述…… 107
4.3.2 圆形窗口的线段裁剪…… 107
4.3.3 任意多边形窗口对圆的裁剪…… 109
4.4 字符裁剪 …… 114

第 5 章 图形变换…… 116

5.1 图形变换的数学基础 …… 116
5.1.1 矢量的定义及运算…… 116
5.1.2 矩阵的定义及运算…… 117
5.1.3 齐次坐标…… 118
5.2 二维图形几何变换 …… 119
5.2.1 二维几何变换概述…… 119
5.2.2 二维图形基本变换…… 120
5.2.3 二维组合变换…… 125
5.2.4 交互技术实现图形变换…… 130

5.3 三维图形几何变换 …… 141
5.3.1 三维图形基本变换及组合变换 …… 141
5.3.2 三维图形的线框拉伸造型方法 …… 146
5.3.3 投影变换 …… 153
5.3.4 三维形体的交互技术 …… 156
5.3.5 透视投影变换 …… 164

第 6 章 消隐技术 …… 170

6.1 消隐相关概念及算法类型 …… 170
6.2 凸多面体的消隐 …… 171
6.2.1 凸、凹多面体的区分 …… 171
6.2.2 利用平面外法线判断可见性 …… 171
6.3 一般多面体的消隐 …… 176
6.3.1 消隐分析 …… 176
6.3.2 隐线算法 …… 176
6.3.3 画家算法 …… 178
6.3.4 深度缓冲器算法 …… 179
6.3.5 基于扫描线的消隐算法 …… 180

第 7 章 真实感图形绘制 …… 210

7.1 相关物理知识 …… 210
7.1.1 基本光学原理 …… 210
7.1.2 颜色与光的关系 …… 211
7.2 光照模型 …… 212
7.2.1 简单光照模型 …… 212
7.2.2 整体光照模型 …… 222
7.3 纹理 …… 226
7.3.1 概述 …… 226
7.3.2 二维纹理映射和三维纹理映射 …… 227

第 8 章 曲线曲面 …… 229

8.1 曲线曲面基础知识 …… 229
8.1.1 曲线和曲面的表示方法 …… 229
8.1.2 连续性、样条及曲线曲面构造方式 …… 230
8.2 Bézier 曲线曲面 …… 231
8.2.1 Bézier 曲线定义 …… 231
8.2.2 Bézier 曲线的性质 …… 232
8.2.3 低次 Bézier 曲线及矩阵表示 …… 233
8.2.4 Bézier 曲线的拼接 …… 234

8.2.5 Bézier 曲线的递推生成算法 …… 235
8.2.6 Bézier 曲面 …… 235
8.3 B样条曲线曲面 …… 236
8.3.1 B样条的一般定义 …… 236
8.3.2 二次和三次B样条曲线段 …… 236
8.3.3 双三次B样条曲面 …… 239
8.3.4 B样条递推定义 …… 239
8.3.5 B样条曲线的类型 …… 240
8.3.6 反求B样条曲线控制点 …… 241
8.3.7 B样条曲线绘制 …… 241
8.3.8 曲面拉伸造型方法 …… 248
8.4 NURBS方法 …… 257

第9章 计算机动画与仿真 …… 260

9.1 计算机动画与仿真的概念及基本原理 …… 260
9.2 计算机动画与仿真的实现方法 …… 261
9.2.1 逐帧动画 …… 261
9.2.2 实时动画 …… 262
9.3 计算机实时动画实践 …… 263

第 二 部 分

第10章 基于OpenGL的图形开发技术 …… 275

10.1 OpenGL开发环境配置及相关规范介绍 …… 275
10.1.1 VC6.0环境OpenGL配置方法 …… 275
10.1.2 OpenGL相关规范介绍 …… 281
10.2 基本图形及真实感图形绘制 …… 282
10.2.1 基本图形绘制 …… 282
10.2.2 图形变换与三维绘图 …… 294
10.2.3 真实感图形显示 …… 301
10.3 OpenGL图像处理技术 …… 319
10.3.1 位图图像 …… 319
10.3.2 像素图像 …… 321
10.3.3 图像操作 …… 324
10.4 OpenGL纹理映射技术 …… 329
10.4.1 纹理映射的一般步骤 …… 329
10.4.2 纹理对象 …… 333
10.4.3 纹理透明 …… 336
10.4.4 一维纹理 …… 337

10.4.5 球体纹理…… 339
10.4.6 立方图纹理及天空盒绘制和表面反射…… 342
10.5 OpenGL 曲线曲面技术…… 350
10.5.1 绘制二次曲面…… 350
10.5.2 绘制 Bézier 曲线曲面…… 353
10.5.3 绘制 NURBS 曲线曲面…… 358
10.5.4 NURBS 曲面修剪…… 364
10.5.5 曲面纹理映射…… 367

第 11 章 Web 图形开发技术…… 369

11.1 Web 绘图技术的结构概述…… 369
11.1.1 HTML 网页文档结构…… 369
11.1.2 JavaScript 概述…… 371
11.1.3 canvas 中的图形…… 372
11.2 Web 环境下基本图形的生成…… 373
11.2.1 直线的绘制…… 374
11.2.2 封闭多边形的绘制…… 375
11.2.3 多边形的扫描填充…… 376
11.2.4 多边形的裁剪…… 378
11.2.5 二维图形的组合变换…… 382
11.2.6 二维图形拉伸生成三维图形…… 391
11.2.7 三维图形的组合变换…… 393
11.3 基于 WebGL 的 3D 图形…… 398
11.3.1 Three.js 绘制 3D 图形的结构…… 399
11.3.2 Web 下的三维模型的显示…… 400

参考文献…… 402

第一部分

第 1 章

绪 论

计算机图形学(computer graphics,CG)是建立在传统图学理论、应用数学和计算机科学等基础上的一门学科,广泛应用于很多领域,计算机辅助设计、计算机动画仿真、虚拟现实和计算机可视化等相关学科和技术都以计算机图形学的原理和算法为基础。虽然现在各学科之间的研究内容相互交叉、相互渗透,使学科界限逐渐模糊,但是计算机图形学仍然具有明显的学科特点。

1.1 概　　念

由于计算机图形学涉及的图形内容很广泛,因此,对其很难有一个统一的定义,国内外各种专业文献对它的概念也有不同的表述。

国际标准化组织(International Organization for Standardization,ISO)给出的定义是:计算机图形学是研究通过计算机将数据转换为图形,并在专门显示设备上显示的原理、方法和技术的学科。

美国电气电子工程师学会(Institute of Electrical and Electronics Engineers,IEEE)给出的定义是:计算机图形学是利用计算机产生图形影像的一门艺术或科学。

目前,国内广泛采用的对计算机图形学的定义是:计算机图形学是研究怎样利用计算机来表示、生成、处理和显示图形的原理、算法、方法和技术的一门学科。随着其应用领域的不断扩展和延伸,人们对计算机图形学的认识会进一步深入。

1.2 研究内容及应用领域

1.2.1 图形和图像的关系

对计算机图形学的研究首先要区分图形和图像两个概念,二者虽然有联系,而且区别也越来越模糊,但还是有区分的。

广义上,能在人的视觉系统中产生视觉印象的客观对象都可以称为图形,包括自然景物、拍摄的照片、用数学方法描述的图形,等等。狭义上,计算机图形学中的图形是指用数学方法描述的形状,图形通常由点、线、面、体等几何元素和灰度、色彩、线型、线宽等非几何属

性组成。从构成要素上看,图形主要分为两类:一类是几何要素在构图中具有突出作用的图形,如工程图、等高线地图、曲面的线框图等;另一类是非几何要素在构图中具有突出作用的图形,如明暗图、晕渲图、真实感图形等,这样的图形又称为矢量图形。例如,一幅花的矢量图形实际上是由线段形成外框轮廓,由外框的颜色以及外框所封闭的颜色决定花显示出的颜色。矢量图只有通过图形软件才能生成,图形文件在计算机的硬盘和内存中占用的空间较小,图形放大或者缩小后图像不会失真,显示质量和显示设备的分辨率无关。

图像是广义上的图形,它是指通过诸如视觉系统看到的一幅景象、照相机拍的一张照片、图像扫描设备扫描获得的图片等方式获得的图形,在计算机内图像以点阵位图(bitmap)的形式呈现,图像中的每一个点记录了图像在该点的灰度、亮度或者颜色值,将图像中所有点的灰度、亮度或者颜色值组合在一起才能得到图像的整体信息。因此,图像需要记录每一个图像点,相对于图形文件,图像文件占用的计算机空间较大,显示会发生失真现象,显示质量和显示设备的分辨率有关。

1.2.2 图形输入输出硬件技术

从软硬件上划分,计算机图形学的研究大致可分为两个方面:一是计算机对图形数据输入、输出的硬件技术研究,二是图形数据的计算、处理和存储的软件技术研究。

由于计算机图形学最初是由于计算机图形硬件的发展而产生的,因此图形硬件是其重要的研究内容之一,例如图形的输入、输出设备和技术,包括显示设备的结构体系,硬件交互、接口等方面。

图形输入设备常用的是键盘和鼠标,其他的还有坐标数字化仪、图形扫描仪、触摸屏、光笔、操纵杆以及数据手套等,三维扫描仪是现在的研究热点之一,它可以通过直接扫描空间物体来获得物体的立体图形数据。图形输入设备获得的图形分为矢量型图形和光栅扫描型图形两种类型,矢量型图形即我们所讲的图形(graphics),记录的是图形的几何要素(轮廓和形状等)以及非几何要素(颜色、材质等),光栅扫描型图形获得的数据是由亮度值构成的像素矩阵——图像(image),图像数据转化为图形数据后,即可用于计算机图形相关软件中。图形输入设备的重要性能指标是图形输入的精度。

图形输出设备包括显示器、打印机、绘图仪等,图形数据经过计算后可在显示器上呈现当前的图像状态或者图形编辑后的结果,也可以通过打印机、绘图仪在纸质介质上保留下来,以便长期保存。当前图形输出设备研究的热点之一——三维打印机,可以将空间立体的图形数据直接快速成型。图形输出设备的重要性能指标是图形输出的精度。

1.2.3 计算机图形学的主要研究内容

由于计算机图形学是研究利用计算机来表示、处理和显示图形的原理、方法和技术的学科,所以,凡是和此相关的内容都是计算机图形学的研究内容。简单来说,从基本图形到复杂图形,从二维图形到三维图形,从静态图形到动态仿真图形,从线框和实体模型到真实感图形、虚拟现实、真实场景以及其他相关计算机图形表示等都属于计算机图形学的

研究范畴。而且，由于计算机技术的迅猛发展，计算机图形学的研究内容也在不断变化和丰富完善。

计算机图形学的研究主要是围绕图形信息的输入、表达、存储、显示、变换以及图形准确性、真实性和实时性的基础算法进行的，其算法可以分为以下几类。

(1) 基于图形设备的基本图形数据结构和图形元素的生成算法，如光栅图形显示器生成直线、圆弧、二次曲线、封闭边界内的图案填充，以及反走样等。

(2) 图形元素的几何变换、投影变换、窗口裁剪等。

(3) 自由曲线和曲面的插值、拟合、拼接、分解、过渡、光顺、整体和局部修改等。

(4) 图形元素(点、线、面、体、环)的求交以及集合运算。

(5) 隐藏线、隐藏面消隐算法以及具有光照模型效果的真实感图形显示算法。

(6) 不同字体的点阵表示。

(7) 山、水、花、烟云等模糊景物的生成算法。

(8) 三维形体的实时显示和处理。

(9) 虚拟现实环境的生成及其控制算法。

除了上述内容外，图形交互技术、图像生成算法、色彩处理、图形操作和处理、图形优化和加速、图形信息的描述和表示、图形数据的存储和检索以及编码等技术也是计算机图形学的研究内容。由于计算机图形学的研究问题来源于日常生活，以及科学、工程技术、艺术、影视、游戏、医疗、军事、教育等领域，因此，计算机图形学的研究目的也是解决相关领域的实际需求，例如，科学计算可视化和三维或高维数据场的可视化技术，可将科学计算中大量难以理解的数据通过计算机图形显示出来，从而使人们加深对其科学过程的理解；有限元分析结果、应力场/磁场的分布、海洋洋流运动、气候变化分布以及各种复杂的运动学和动力学问题可以通过图形仿真来直观地呈现；除此以外，计算机动画、自然景物仿真、虚拟现实、地理信息系统等也属于计算机图形学的研究内容。而且由于学科交叉和融合，很多研究内容和技术已经从计算机图形学中独立出来，成为一门新的学科。

为使读者理解和掌握计算机图形学的基本理论和方法，本书着重讨论如何在计算机中表示图形，以及如何利用计算机进行图形的生成、处理和显示的相关原理与算法，为进一步学习和研究计算机图形学的相关问题打下坚实的基础。本书对计算机图形学的研究主要集中在以下方面：

(1) 图形生成技术研究，包括线段、圆弧、字符、区域填充、消隐、光照模型、纹理、灰度与色彩等各种真实感图形生成技术；

(2) 几何模型构造技术研究，包括二维/三维几何模型、自由曲线和曲面造型等；

(3) 图形编辑与处理技术研究，包括图形的平移、旋转、缩放、投影、裁剪等几何变换，三维几何投影变换；

(4) 动画技术；

(5) 图形动态交互拾取技术；

(6) OpenGL 图形开发技术；

(7) Web 环境下图形开发技术。

1.2.4 计算机图形学的应用领域

从相关研究内容可以看出，以计算机图形为基础的相关软硬件技术已经广泛应用于很多领域，如科学、工程、医药、工业、艺术娱乐、广告和教学等，直接或者间接地对我们的工作、学习和生活产生了深刻的影响。

计算机辅助设计与制造(CAD/CAM)是计算机图形学最广泛、最活跃的应用领域。利用计算机图形学的基本原理和方法研发的CAD/CAM软件，已广泛地应用于机械、建筑等产品和工程的设计，如飞机、汽车、船舶、建筑、轻工、机电、服装的外形设计，大规模集成电路、电子器件的设计以及工厂企业的布局等。CAD软件现在已经是工程产品设计必不可少的工具，它可以极大缩短产品设计周期，节省原材料，提高产品设计质量等，其产生的经济效益十分明显。CAD软件中的三维几何造型技术具有很多优点，除了造型便捷外，还可以进行装配件的虚拟装配、干涉检查等，结合CAM和CAE(计算机辅助工程分析)等软件或功能模块，对产品进行仿真数控加工、有限元分析等，基本上代表了CAD的发展方向。现在产品设计已不再是一个设计领域内孤立的技术问题，而是综合了产品各个相关领域、相关工程、相关技术资源和相关组织形式的系统化工程。在网络环境下进行异地异构系统的协同设计，已成为CAD领域的研究热点之一。

科研、工程、商业及社会中的各个行业都会产生大量的数据，从这些"数据海洋"中提取有价值的信息，并通过数据分析和处理找到变化的规律及数据反映的本质特征尤为重要，以计算机图形学为基础的科学计算的可视化技术将数据转化为图形或者图像显示出来，而且根据需要也可以进行交互处理，对数据处理非常有帮助。1987年2月英国国家科学基金会在华盛顿召开了有关科学计算可视化的首次会议，会议一致认为"将图形和图像技术应用于科学计算是一个全新的领域"，科学家们不仅需要分析由计算机得出的计算数据，而且需要了解在计算过程中数据的变化。会议将这一技术定名为"科学计算可视化"(visualization in scientific computing)。科学计算可视化将图形生成技术、图像理解技术结合在一起，它既可理解送入计算机的图像数据，也可以从复杂的多维数据中产生图形。它涉及下列相互独立的几个领域：计算机图形学、图像处理、计算机视觉、计算机辅助设计及交互技术等。科学计算可视化按其实现的功能来分，可以分为三个档次：①结果数据的后处理；②结果数据的实时跟踪处理及显示；③结果数据的实时显示及交互处理。科学计算可视化技术根据所研究对象的领域的不同，可分为科学可视化、数据可视化和信息可视化。由于社会活动日益频繁，数据量呈爆炸式的增加，可视化技术有着广阔的发展前途。

虚拟现实技术是近几年的研究重点之一，"虚拟现实"(virtual reality)一词是由美国喷气推动实验室(VPL)的创始人拉尼尔(Jaron Lanier)首先提出的，在克鲁格(Myren Kruege)20世纪70年代中早期的实验中被称为"人工现实"(artificial reality)，而在吉布森(William Gibson)1984年出版的科幻小说*Neuremanccr*中，又被称为"可控空间"(cyber space)。简单来说，虚拟现实技术就是人们利用计算机生成一个逼真的三维虚拟环境，通过自然动作操作传感设备来与之相互作用的新技术。与传统的数字仿真系统相比，利用虚拟现实技术构造出来的可视化虚拟现实系统具有三个重要特征：一是沉浸性，体验者的确有"看得见、摸得着、听得到、闻得出"的身临其境的真实感受；二是交互性，体验者使用日常生

活中的方式与虚拟场景中的人或物进行各种交流，产生真实的互动体会；三是构想性，用户在虚拟的环境中获取新的知识和体验，形成感性或理性的认识，从而产生新的思想和行动，有效提高思考和行动能力。虚拟现实技术主要研究用计算机模拟（构造）三维图形空间，并使用户能够自然地与该空间进行交互。它涉及很多学科的知识，对三维图形处理技术的要求特别高。简单的虚拟现实系统早在20世纪70年代便被应用于军事领域，用来训练驾驶员。80年代后，随着计算机软硬件技术的提高，该系统也得到重视并迅速发展。目前，它已在航空航天、医学、教育、艺术、建筑等领域得到初步的应用。例如，1997年7月，美国国家航空航天局的“旅居者号”火星车着陆距地球约1.9亿km的火星。这辆在火星表面缓慢爬行的小车中并没有驾驶员，它是由地球上的工程师通过虚拟现实系统操纵的。虚拟现实技术的应用范围很广，例如用于脑外科规划的双手操作空间接口工具。美国弗吉尼亚大学推出了一种能用于脑外科规划的被称为Netra的双手操作空间接口工具，根据脑外科医生的工作环境和习惯，该系统采用一种外形像人头的控制器。脑外科医生可以根据他们的职业习惯，通过转动外形像人头的控制器来方便地观察人脑的不同部位，同时通过右手控制面板的平面来控制人脑的剖面的扫描，并能根据CT或强磁共振图像所产生的主体脑模型显示所需得到观察视点着色后的真实图像。虚拟环境也可用于恐高症治疗、虚拟风洞实验，作为封闭式战斗作战训练器，以及用于建筑设计中。

地理信息系统（geographical information system，GIS）是建立在地理图形之上的关于人口、矿藏、森林、旅游等资源的综合信息管理系统。在地理信息系统中，计算机图形学技术被用来产生高精度的各种资源的图形，包括地理图、地形图、森林分布图、人口分布图、矿藏分布图、气象图、水资源分布图等。地理信息系统可以为管理和决策者提供非常有效的支持，它在发达国家中已得到广泛应用，我国也对其开展了广泛的研究与应用。例如数字化地图是地理信息系统在人们日常生活中的一个直接应用，给人们的旅游等带来了极大的便利。

除了上述领域外，计算机图形学还应用于软件的交互界面设计、计算机动画、影视特效和游戏制作等。归纳起来，计算机图形学可应用于如下学科和领域：

(1) 计算生物学(computational biology)
(2) 计算物理学(computational physics)
(3) 计算机辅助设计(computer aided design)
(4) 数字化艺术(digital art)
(5) 教育(education)
(6) 图形设计(graphic design)
(7) 信息几何(infographics)
(8) 信息可视化(information visualization)
(9) 理论药物设计(rational drug design)
(10) 交通可视化(scientific visualization)
(11) 视频游戏(video games)
(12) 虚拟现实(virtual reality)
(13) 网络设计(web design)

1.3 发展历史

计算机图形学是伴随着计算机的出现而产生的，主要经历了以下几个发展阶段。

1.3.1 萌芽阶段

世界上第一台数字电子通用计算机 ENIAC 于 1946 年 2 月 14 日在美国宾夕法尼亚大学研制成功，由于没有连接图形显示设备，因此，这时的计算机和图形学之间没有建立联系。1950 年，美国麻省理工学院(MIT)“旋风 1 号”计算机(whirlwind 1)配备了一个图形显示器，该显示器用一个类似于示波器的阴极射线管(CRT)来显示一些简单的图形，CRT 的出现为计算机生成和显示图形提供了可能。1958 年美国 Calcomp 公司将联机的数字记录仪发展成滚筒式绘图仪，GerBer 公司把数控机床发展成平板式绘图仪。由于当时的计算机主要应用于科学计算，为这些计算机配置的图形设备仅具有输出功能，不具备人机交互功能，计算机图形学在这个阶段尚处于准备和酝酿时期。到 20 世纪 50 年代末期，MIT 的林肯实验室在“旋风”计算机上开发了 SAGE 空中防御体系，第一次使用了具有指挥和控制功能的 CRT 显示器，操作者可以用光笔在屏幕上指出被确定的目标。与此同时，类似的技术在设计和生产过程中也陆续得到了应用，这是交互式计算机图形系统的雏形，预示着交互式计算机图形技术的诞生。

1.3.2 发展阶段

1962 年，MIT 林肯实验室的 Ivan E. Sutherland 发表了一篇题为“Sketchpad：一个人机交互通信的图形系统”的博士论文，他在论文中首次使用了计算机图形学(computer graphics)这个术语，证明了交互计算机图形学是一个可行的、有用的研究领域，从而确定了计算机图形学作为一个崭新的学科分支的独立地位；他在论文中所提出的一些基本概念和技术，如交互技术、分层存储符号的数据结构等至今还在广为应用。1964 年 MIT 的教授 Steven A. Coons 提出了被后人称为超限插值的新思想，通过插值四条任意的边界曲线来构造曲面。同在 20 世纪 60 年代早期，法国雷诺汽车公司的工程师 Pierre Bézier 发展了一套被后人称为 Bézier 曲线、曲面的理论，成功地用于几何外形设计，并开发了用于汽车外形设计的 UNISURF 系统。Coons 方法和 Bézier 方法是 CAGD 最早的开创性工作。值得一提的是，计算机图形学的最高奖是以 Coons 的名字命名的，而获得第一届(1983 年)和第二届(1985 年) Steven A. Coons 奖的，恰好是 Ivan E. Sutherland 和 Pierre Bézier。Ivan E. Sutherland 被称为“计算机图形学之父”，获得了 1988 年的计算机界的最高奖“图灵奖”和 IEEE 计算机杰出成就奖，他也

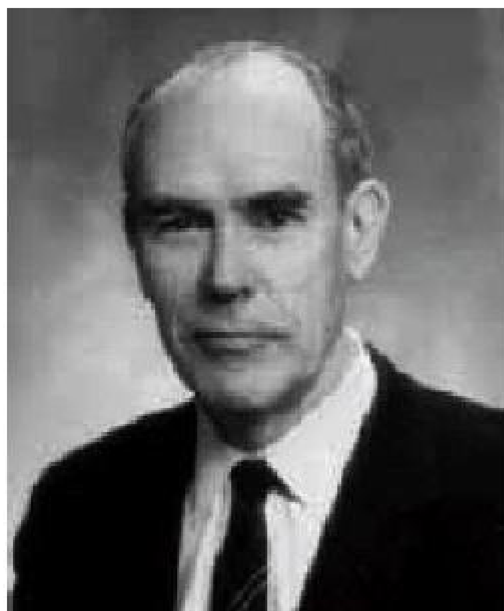


图 1.3-1 计算机图形学学科创始人——Ivan E. Sutherland

是许多图形学基本算法的创始人(见图 1.3-1)。

1.3.3 推广应用阶段

20 世纪 70 年代是计算机图形学发展过程中一个重要的历史时期。由于光栅显示器的产生,在 60 年代就已萌芽的光栅图形学算法迅速发展起来,区域填充、裁剪、消隐等基本图形概念及其相应算法纷纷诞生,图形学进入了第一个兴盛的时期,并开始出现实用的 CAD 图形系统。又因为通用、与设备无关的图形软件的发展,图形软件功能的标准化问题被提了出来。1974 年,美国国家标准化局(American National Standards Institute,ANSI)在 ACM SIGGRAPH 的一个"与机器无关的图形技术"的工作会议上,提出了制定有关标准的基本规则。此后 ACM 专门成立了一个图形标准化委员会,开始制定有关标准。该委员会于 1977 年、1979 年先后制定和修改了"核心图形系统"(core graphics system)。ISO 随后又发布了计算机图形接口(computer graphics interface,CGI)、计算机图形元文件标准(computer graphics metafile,CGM)、计算机图形核心系统(graphics kernel system,GKS)、面向程序员的层次交互图形标准(programmer's hierarchical interactive graphics standard,PHIGS)等。这些标准的制定,对计算机图形学的推广、应用以及资源信息共享起到了重要作用。

在 20 世纪 70 年代,计算机图形学的另外两个重要进展是真实感图形学和实体造型技术的产生。1970 年 Bouknight 提出了第一个光反射模型,1971 年 Gourand 提出"漫反射模型+插值"的思想,被称为 Gourand 明暗处理。1975 年 Phong 提出了著名的简单光照模型——Phong 模型。这些可以算是真实感图形学最早的开创性工作。另外,从 1973 年开始,相继出现了英国剑桥大学 CAD 小组的 Build 系统、美国罗彻斯特大学的 PADL-1 系统等实体造型系统。

1.3.4 实用化阶段

1980 年 Whitted 提出了一个光透视模型——Whitted 模型,并第一次给出光线跟踪算法的范例——实现 Whitted 模型;1984 年,美国 Cornell 大学和日本广岛大学的学者分别将热辐射工程中的辐射度方法引入计算机图形学中,用辐射度方法成功地模拟了理想漫反射表面间的多重漫反射效果;光线跟踪算法和辐射度算法的提出,标志着真实感图形的显示算法已逐渐成熟。

20 世纪 80 年代以后,工作站的出现也极大促进了计算机图形学的发展。相对小型计算机来说,工作站是一个用户使用一台计算机,交互响应时间短,而且可以共享资源,如大容量磁盘、高精度绘图仪等,在图形生成上具有显著的优点。因此工作站取代小型计算机成为图形生成的主要环境。到了 80 年代后期,微型计算机的性能迅速提高,并配有高分辨率的显示器和窗口管理系统,可在网络环境下运行。由于其价格便宜,因此得到了广泛的推广,尤其是微型计算机上的图形软件和支持图形应用的操作系统及程序(如 Windows、Office、AutoCAD、CoreDraw、FreeHand、3DMax 等)的全面出现,使计算机图形学技术的应用深度和广度得到了前所未有的发展。

1.3.5 标准化、智能化阶段

20世纪90年代，随着多媒体技术的提出，计算机图形学的功能有了很大的提高，计算机图形系统已成为计算机系统必不可少的组成部分。随着面向对象的程序设计语言的发展，出现了面向对象的计算机图形系统，计算机图形学开始朝着标准化、集成化和智能化的方向发展，国际标准化组织公布的图形标准越来越多，且更加成熟，并得到了广泛的认同和采用。这些图形标准包括计算机图形接口（CGI）标准、计算机图形元文件（CGM）标准、图形核心系统（GKS）、三维图形核心系统（GKS-3D）和程序员层次交互图形系统（PHIGS）。图形软件标准制定的主要目标是提供计算机图形操作所需要的功能，包括图形的输入输出、图形数据的组织和交互等，使现有的计算机和图形设备的功能得到有效利用，以满足实际应用的需要，并在不同的计算机系统、不同的应用系统、不同的用户之间进行信息交换，使图形、程序能够重复使用，与设备无关，实现设备的独立性和便于移植，减少应用系统的开发费用和重复开发等。

超大规模集成电路的发展也为图形学的飞速发展奠定了物质基础。计算机的运算能力的提高，图形处理速度的加快，使得图形学的各个研究方向得到充分发展，图形学已广泛应用于动画、科学计算可视化、CAD/CAM、影视娱乐等各个领域。

1.3.6 多学科融合发展阶段

进入21世纪以后，计算机图形学的研究逐渐朝着多学科交叉融合的方向发展，既有与认知计算、机器学习、人机交互的融合，也有与大数据分析、可视化的融合；不仅针对三维数字模型，还涵盖了图像视频，体现出与计算机视觉的深度交叉。计算机图形学快速发展，一个潜在的趋势是不再有明确清晰的主题，而更多地体现出方法和技巧的创新。前沿热点领域包括3D打印、机器人、认知计算、大数据分析与可视化以及虚拟现实技术等。以3D打印为例，它涉及机械制造、材料设计、几何造型、颜色、力学特性等多个学科的交叉，其中与计算机图形学有关的研究内容包括面向3D打印的几何模型高效表示方法、表面效果定制（纹理、颜色）和模型结构优化分析方法等。计算机图形学的研究人员也积极参与到机器人的研究热潮中，除了机器人路径规划和人形机器人运动仿真这些传统的计算机图形学研究内容外，研究人员已经开发了多项机器人在图形学中的应用，例如机器人模仿人类的面部表情，模块化机器人为模块化家具提供支持等。虚拟现实技术的需求也推动了三维复杂环境的实时动态显示技术发展，人体动画技术研究向多方面发展，曲线曲面技术仍然是研究的热点。

1.4 学科发展

从计算机图形学学科发展来看，有以下几个发展趋势。

（1）与图形硬件的发展紧密结合，突破实时高真实感、高分辨率渲染的技术难点。

图形渲染是整个图形学发展的核心。在计算机辅助设计方面，影视动漫以及各类可视

化应用中都对图形渲染结果的高真实感提出了很高的要求。同时,由于显示设备的快速发展,人们要求能提供高清分辨率(1920×1080),进一步要能达到数字电影所能播放的4K分辨率(4096×2060);色彩的动态范围也希望从原来每个通道的8b提高到10b及以上。虽然已有的图形学方法已经能较为真实地再现各类视觉效果,然而为了能提供高分辨率、高动态的渲染效果,必须消耗非常可观的计算能力。一帧精美的高清分辨率图像,单机渲染往往需要耗费数小时至数十小时。为此,传统方法主要采用分布式系统,将渲染任务分配到集群渲染节点中。即使这样,也需要使用上千台计算机,耗费数月时间才能完成一部标准90min长度的影片渲染。

基于图形处理器(graphics processing unit,GPU)的硬件技术得以发展迅速,已经能在一个GPU芯片上采用64nm工艺集成上千个采用单指令多数据流架构的通用计算核心。2009年底,主流图形硬件商nVidia和AMD以及Intel还推出了基于多指令、多数据流计算核心的GPU芯片用于图形加速绘制,以支持DirectX 11及OpenGL 3.0图形标准。最新的图形学研究采用GPU技术,可以充分利用计算指令和数据的并行性,已可在单个工作站上实现百倍于基于CPU方法的渲染速度。

然而已知的实现方法,其实现效果还较为初步,无法实现复杂的视觉特效,离实时的高真实感渲染还有很大差距。其主要原因是:①缺乏良好的数据组织方法。基于GPU的方法由于硬件的架构原因,其数据组织无法如同CPU方法一样组织,因此对复杂的数据结构仍无法提供很好的支持。②缺乏标准高效的GPU高层编程语言、编译器以及相应调试工具。③由于以上两个问题,无法完整地实现适于电影渲染制作的RenderMan标准,以及其他各类基于物理真实感的渲染算法。因此,如何充分利用GPU的计算特性,结合分布式的集群技术解决以上这些难题,从而构造低功耗的渲染服务,是图形学的未来发展趋势之一。

(2) 研究和谐自然的三维模型建模方法。

三维模型建模方法是计算机图形学的重要基础,是生成精美的三维场景和逼真动态效果的前提。然而,传统的三维模型方法,由于其主要思想方法来源于CAD中基于参数式调整的形状构造方法,建模效率低而学习门槛高,不易于普及和让非专业用户使用。而随着计算机图形技术的普及和发展,各类用户都提出了高效的三维建模需求,因此研究和谐自然的三维建模方法是发展的一个重要趋势。

采用合适的交互手段来进行三维模型的快速构造,特别是应用于概念设计和建筑设计领域,已引起了国际同行的广泛关注。由于笔式或草图交互方式非常符合人类原有日常生活中的思考习惯,因此是研究的重点问题。其难点是根据具体的应用领域,与视觉方法相融合,如何设计合理的交互语汇以及对应的过程式“识别—构造”方法。

与此相关的一个问题是基于规则的过程式建模方法。由于Google Earth等数字地图信息系统的广泛应用,对于地图之上的建筑物信息等存在迫切需求,为此,研究者希望通过激光扫描或者视频等获取方式获得相关信息后能迅速地重建出相关三维模型信息。然而单纯的重建方式存在精度低、稳定性差和运算量大等不足,远不能满足实际的需求。因此,最近的研究中,倾向于采用基于规则的过程式建模方法来尝试高效地构造出三维建筑模型,以及相关的树木等结构化场景。

三维建模方法中的另一主要问题是研究合适的曲面表达方法,以适于各类图形学的应用。在CAD中的主流方法是采用非均匀有理B-样条方法(nonuniform rational B-spline,NURBS),

然而此类方法无法很好地解决非正规情况下的曲面拼合，不太适合于图形学。为此，细分曲面方法作为一种离散迭代的曲面构造方法，由于其构造过程朴素简单以及实现容易，因而成为一个方兴未艾的研究热点，而且极有可能逐步取代 NURBS 方法。主要需要解决的问题有：①奇异点处的 C 连续性的有效构造方法；②与 GPU 图形硬件相结合的曲面处理方法。

（3）利用日益增长的计算性能，实现具有高度物理真实的动态仿真。

高度物理真实感的动态模拟，包括对各种形变、水、气、云、烟雾、燃烧、爆炸、撕裂、老化等物理现象的真实模拟，是计算机图形学一直试图达到的目标。这一技术是各类动态仿真应用的核心技术，可以极大地提高虚拟现实系统的沉浸感。然而高度物理真实性模拟，主要受限于计算机的处理能力和存储容量，不能处理很高精度的模拟，也无法做到很高的响应速度。所幸的是，GPU 技术带来了革新这一技术的可能。充分利用 GPU 硬件内部的并行性，研究者开始普遍关注基于 GPU 的各类数学物理方程求解及其相关的有限元加速计算方法，主要研究关注的焦点还是单个物理方法的 GPU 实现。然而，随着 nVidia 推出了基于 GPU 的 PhysX 通用物理加速技术，以及 Havok 公司与 AMD 合作开发了通用物理中间件技术，相信未来可为高度物理真实的动态模拟提供新的研究机遇。

（4）研究多种高精度数据获取与处理技术，增强图形技术的表现。

为获得真实感的画面与逼真动态效果，一种有效的解决途径是采用各种高精度手段获取所需的几何、纹理以及动态信息。为此，研究者正在考虑对各个尺度上的信息进行获取：小到物体表面的微结构、纹理属性和反射属性通过研制特殊装置予以捕获与处理，或采用一组摄像机来获取演员的几何形体与动态；大到采用激光扫描获取整幢建筑物的三维数据。这里需要研究的三个问题是：①图形获取设备的设计与实现，这是与计算机视觉、硬件、软件相关的系统工程研究问题；②由于一般获取的数据均极为庞大且附加了各种噪声与冗余信息，如何进行处理与压缩以适合于图形学应用；③一旦获取相关的数据，如何进行重用。因此使得基于数据驱动的方法、与机器学习相交叉的图形学方法成为最近的研究热点。

（5）计算机图形学与图像视频处理技术的结合。

家用数字相机和摄像机的日益普及，使得对于数字图像与视频数据的处理成为了计算机研究中的热点问题。而计算机图形学技术恰可以与这些图像处理、视觉方法交叉融合，来直接生成风格化的画面，实现基于图像三维建模，以及直接基于视频和图像数据来生成动画序列。计算机图形学正向的图像生成方法和计算机视觉中逆向地从图像中恢复各种信息方法相结合，可以带来无可限量的想象空间，构造出很多视觉特效来，最终用于增强现实、数字地图、虚拟博物馆展示等。

（6）从追求绝对的真实感向追求与强调图形的表意性转变。

计算机图形学在追求真实感方向的研究发展已进入一个发展的平台期，基本上各种真实感特效在不计较计算代价的前提下均能较好得以重现。然而，人们创造和生成图片的终极目的不仅是展现真实的世界，更重要的是表达所需要传达的信息。例如，在一个需要描绘的场景中每个对象和元素都有其相关需要传达的信息，可根据重要度不同采用不同的绘制策略来进行分层渲染再加以融合，最终合成具有一定表意性的图像。为此，研究者已经开始研究如何将图像处理、人工智能、心理认知等领域相结合，探索合适的表意性图形生成方法。而这一技术趋势的兴起，实际上延续了已有的非真实感绘制研究中的若干进展，必将在未来有更多的发展。

第 2 章

图形开发工具及使用

为了实现计算机图形学的基本理论和图形算法，需要首先搭建图形的开发平台。由于图形可视化界面已经是计算机系统必不可少的组成要素，因此，现在几乎所有的计算机开发语言都具有图形开发功能，例如早期的 C 语言，以及 Basic、C++、Java 等语言均提供相关的图形编程函数，近期的 C＃、JavaScript 脚本语言等也都支持图形编程。计算机图形学理论早期常采用 Turbo C 实现，随着美国微软公司的 Windows 系统在微型计算机领域中的普及使用，微软提供的 Visual Studio 集成开发环境成为应用程序开发的主流平台。其中 Visual C++是 Visual Studio 集成开发平台中广泛使用的程序开发框架，Visual C++采用 C++语言开发程序，可以建立、调试和发布 Windows 应用程序，并具有可视化的类和函数，界面友好、操作简便，极大简化了程序的开发过程。Visual C++除了开发常用的应用程序外，也非常适合计算机图形学的交互式图形开发，是一个理想的计算机图形学理论和算法的编程工具。由于互联网技术高速发展，移动设备迅速普及，在新应用环境下支持计算机图形学的开发平台也在逐渐形成，如 HTML5 图形标准开发等。由于本书的目的是理解和掌握计算机图形学的理论和方法，希望有一个相对成熟的图形开发环境，所以，本书仍然采用 Visual C++(简称 VC++)作为图形开发框架，具体使用经典的 Visual C++ 6.0(简称 VC6.0)版本，在该版本下所开发的代码在 Visual Studio 更高级系列版本中仍可使用。

2.1　VC++开发系统简介

2.1.1　VC6.0 系统介绍

VC6.0 是一个可视化的软件开发框架，利用其中的应用程序创建向导功能，可以帮助快速创建图形软件。在 Windows 系统中安装 VC6.0 后，桌面上会出现 VC6.0 的快捷方式，打开后软件界面如图 2.1-1 所示。

为了快速创建软件，可以使用 VC6.0 的应用程序创建向导来实现。在菜单栏中选择“文件”→“新建”命令，打开“新建”对话框，如图 2.1-2 所示。

在“工程”选项卡中，选中 MFC AppWizard(exe)选项，在“工程名称”文本框中输入要创建的程序的名字，例如 CGTest001，单击“确定”按钮，VC6.0 的创建向导开始自动创建应用程序，并弹出一系列的对话框进行程序设置。其中，在步骤 1 的对话框中，在“您要创建的应用程序类型是：”的选项中，选择“单文档”单选按钮，然后单击“下一步”按钮，进行下一步的

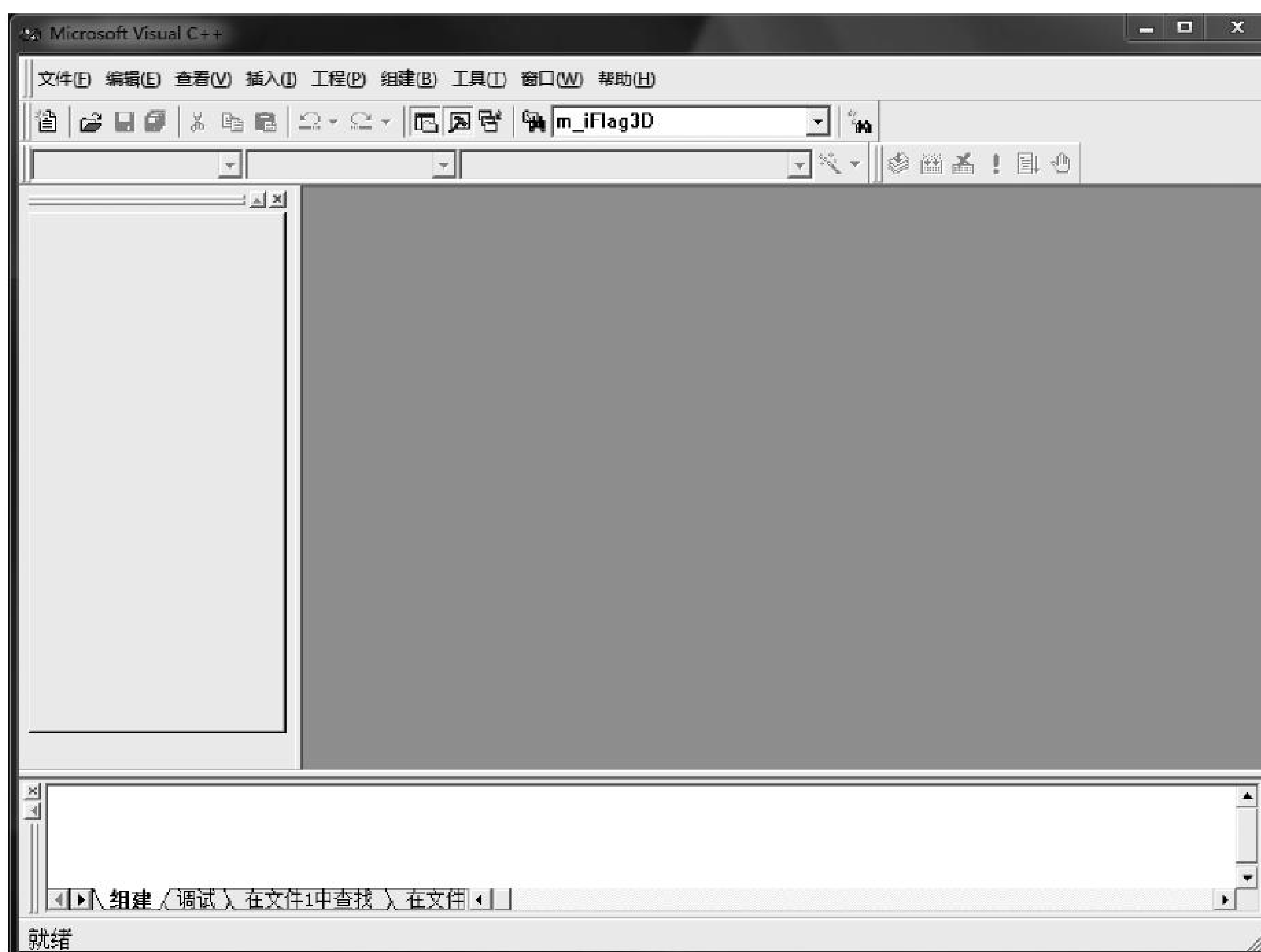


图 2.1-1 VC6.0 界面

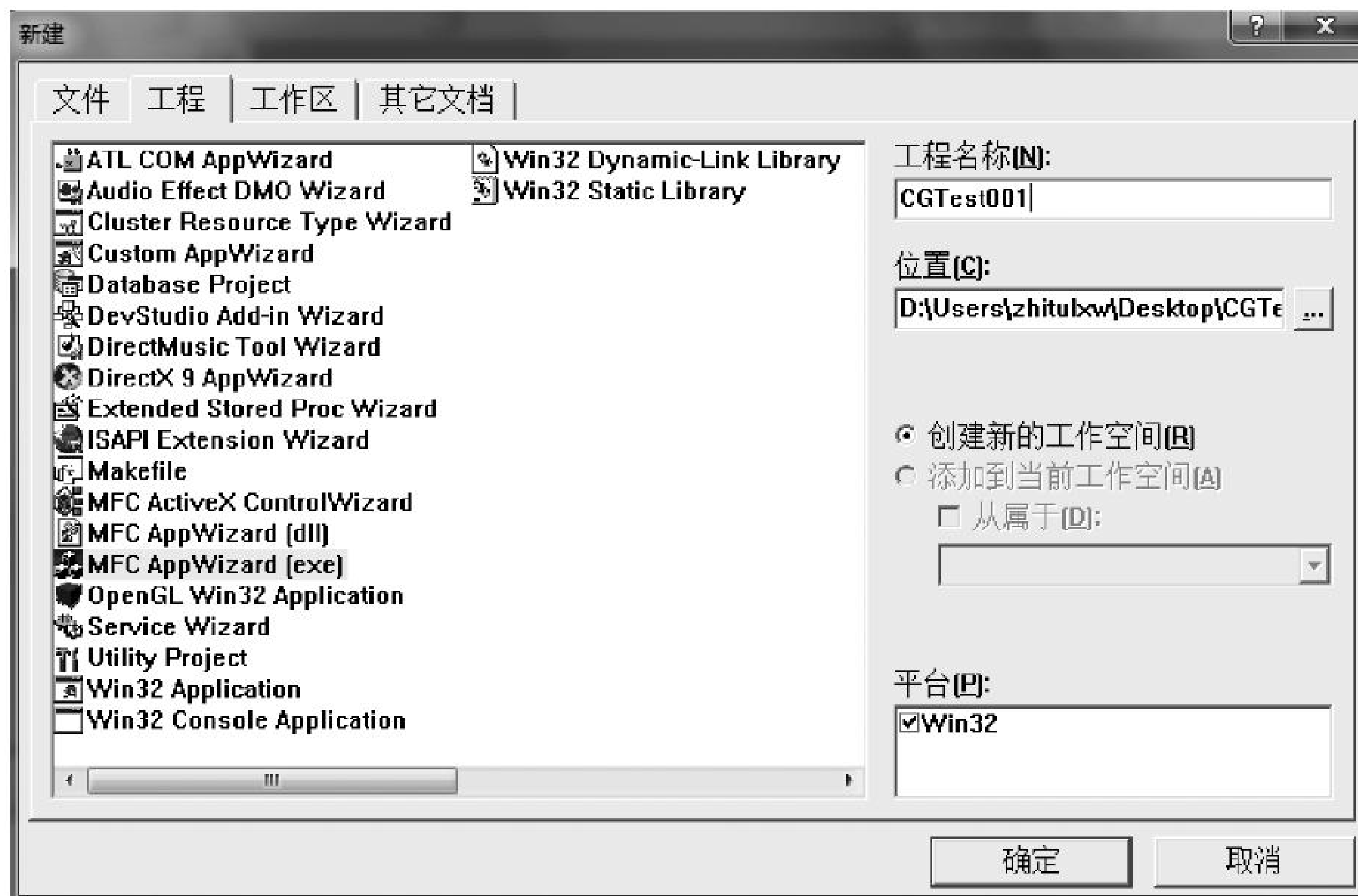


图 2.1-2 "新建"对话框

设置，如图 2.1-3 所示。

在应用程序向导创建的其他步骤中，接受默认的选项，并单击“下一步”按钮，在最后一步，可以看到向导为应用程序自动创建的类及类文件，如图 2.1-4 所示。



图 2.1-3　程序创建向导步骤 1



图 2.1-4　程序创建向导步骤 6

创建好的应用程序开发平台如图 2.1-5 所示。在左侧的工作空间窗口中，可以看到已经创建的程序框架对应的类、资源和文件。

在菜单栏中选择“组建”→“执行[CGTest001.exe](Ctrl+F5)”命令，编译并打开创建好的应用程序，软件界面如图 2.1-6 所示。这是一个标准的 Windows 风格的应用程序，上面有菜单和默认的工具栏，中间的空白区域即为绘图区域(或者称为视图窗口)。

软件界面中的所有内容，例如菜单、工具条、状态栏、对话框以及绘图窗口等，在应用程序中都有对应的资源、类和类文件，而且是一一对应的关系。其中绘图区域对应的是应用程序中

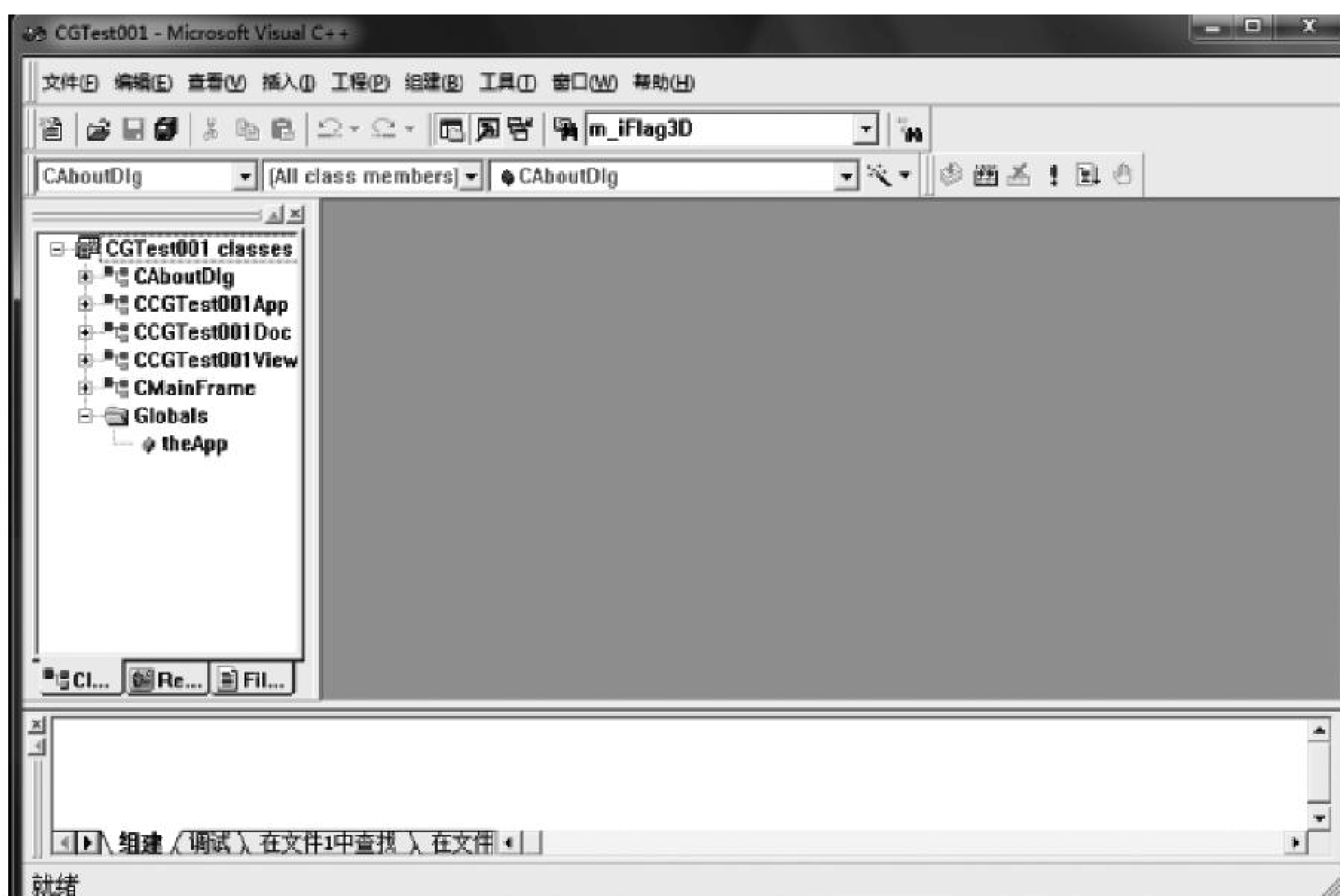


图 2.1-5　程序框架

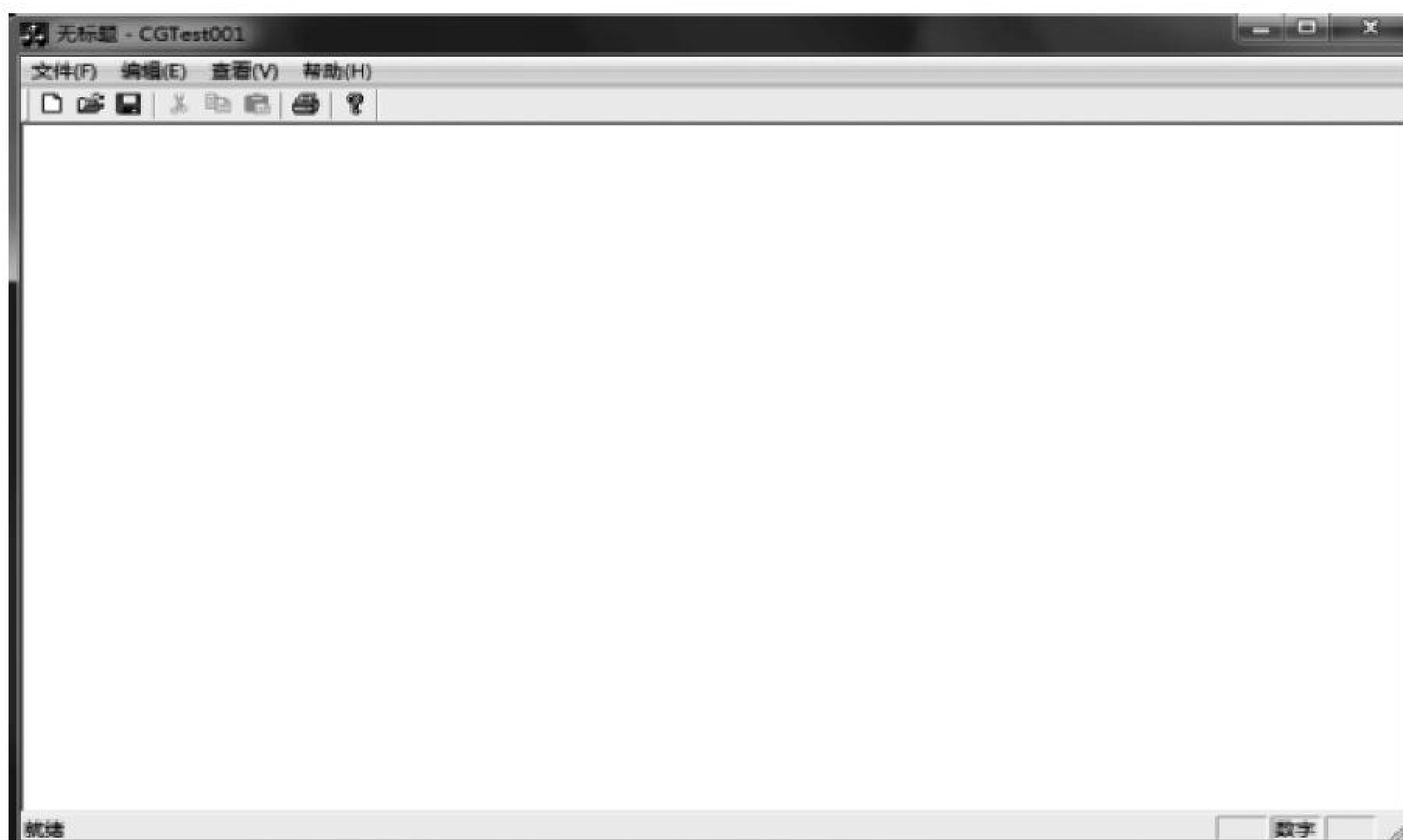


图 2.1-6　程序软件界面

的视图类，图示中的视图类是 CCGTest001View，对应的文件是头文件 CGTest001View.h 和源文件 CGTest001View.cpp，如图 2.1-7 所示。

应用程序中其他的类及文件，如 CAboutDlg、CCGTest001App、CCGTest001Doc 和 CMainFrame 等，是创建软件程序框架必须具备的部分，但在实现计算机图形原理算法时用不到这些类及文件，不用考虑其实际意义。

在默认状态下，视图窗口的左上角是绘图区域坐标系的原点，X 轴向右，Y 轴向下，如图 2.1-8 所示。在实现计算机图形学的算法时，以显示器的像素点作为长度单位。

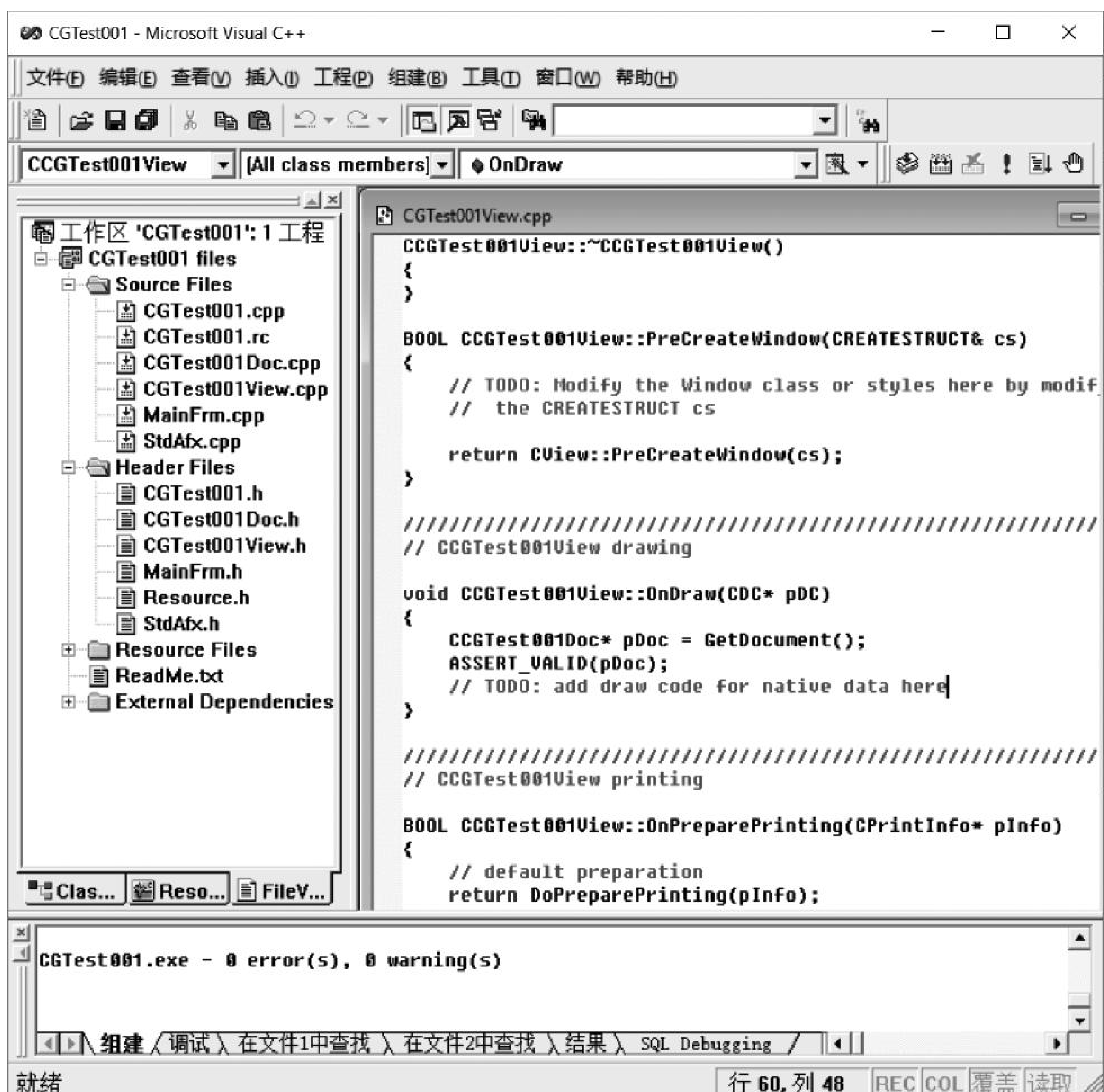


图 2.1-7　视图类 CCGTest001View

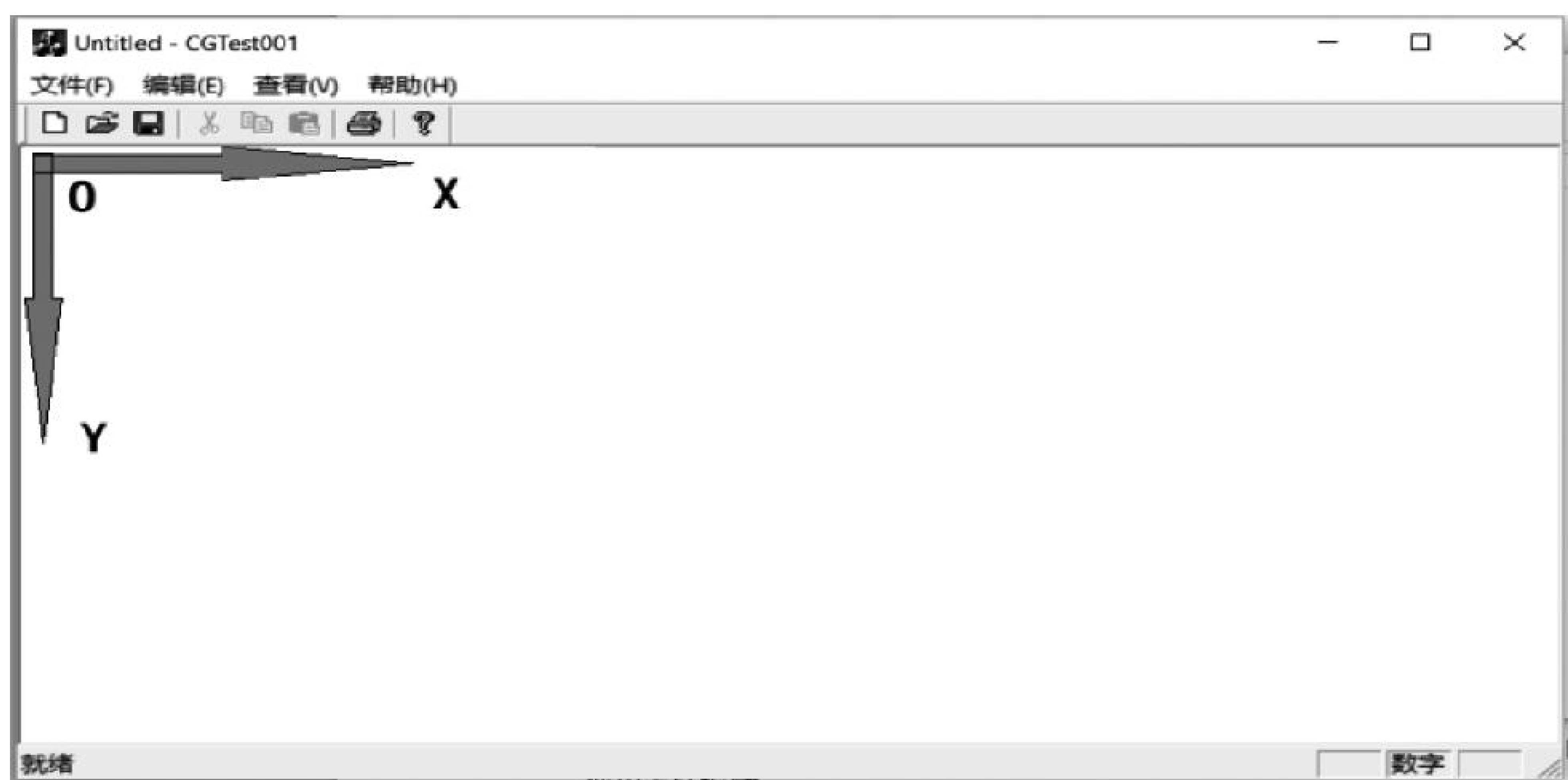


图 2.1-8　绘图区域坐标系

2.1.2　VC++相关设计流程

在 VC6.0 中，可视化窗口上的任何操作，例如，单击菜单里的菜单项、单击工具栏里的工具图标以及在绘图区域单击、右击、移动鼠标或者双击等，都会被视为一种消息(Message)，如果需要对该消息进行相应的响应，则进入对应的消息函数中进行处理。例如，在绘图区域的任意位置单击，就会触发视图类 CCGTest001View 中的 WM_LBUTTONUP 这个消息，如果对该消息进行响应，则进入对应的消息函数 OnLbuttonUp()中进行相应处理，上述过程的消息函数操作如图 2.1-9 所示。在开发环境的菜单栏中选择“查看”→“建立类向导”命令，打开类向导(MFC ClassWizard)对话框，切换到消息映射(Message Maps)选项卡，在类名(Class Name)下拉列表框中选择视图类(CCGTest001View)，对象标识(Object IDs)列表框中选择视图类名本身，在消息(Messages)列表框中选择 WM_LBTTONUP 选项，双击或者单击 Add Function 按钮，在视图类里增加鼠标左键单击的消息函数 OnLbuttonUp()，在类头文件 CGTest001View.h 中可以看到声明的该函数名，在源文件 CGTest001View.cpp 中可以看到该函数体。

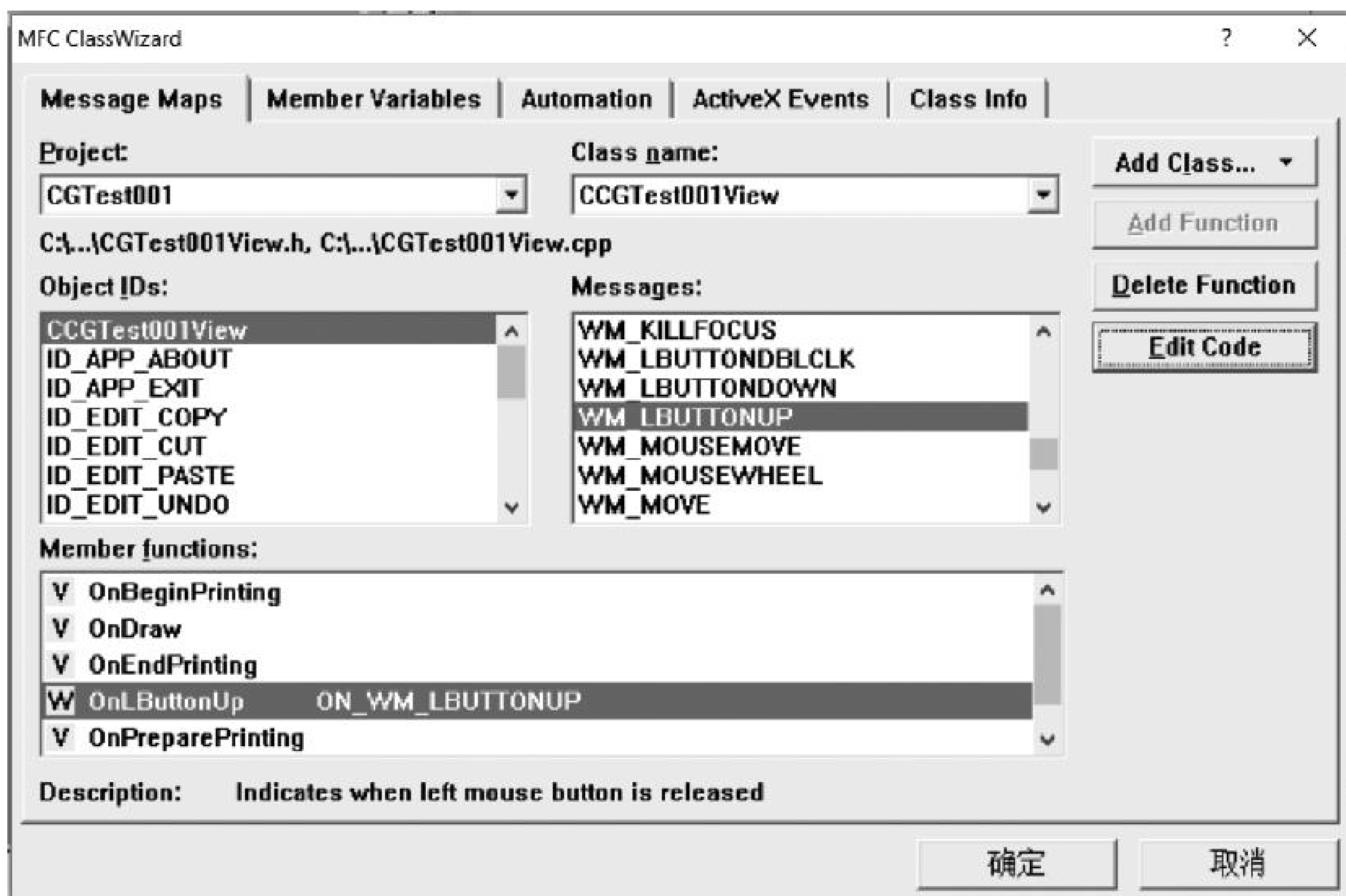


图 2.1-9　声明消息函数

假如，要求单击时弹出一个对话框，则 OnLButtonUp()函数中的代码为：

```
void CCGTest001View::OnLButtonUp(UINT nFlags, CPoint point) {
    //TODO: Add your message handler code here and/or call default
    MessageBox("现在是鼠标左键点击!");        //本行为增加的代码
    CView::OnLButtonUp(nFlags, point);
}
```

编译并运行增加代码后的应用程序，打开软件，在绘图区域单击，弹出窗口，如图 2.1-10 所示。鼠标其他操作的函数设置和上述方法类似。

VC6.0 中的所有资源(菜单、工具栏、对话框、图标等)都有唯一的标识符号，可以在对

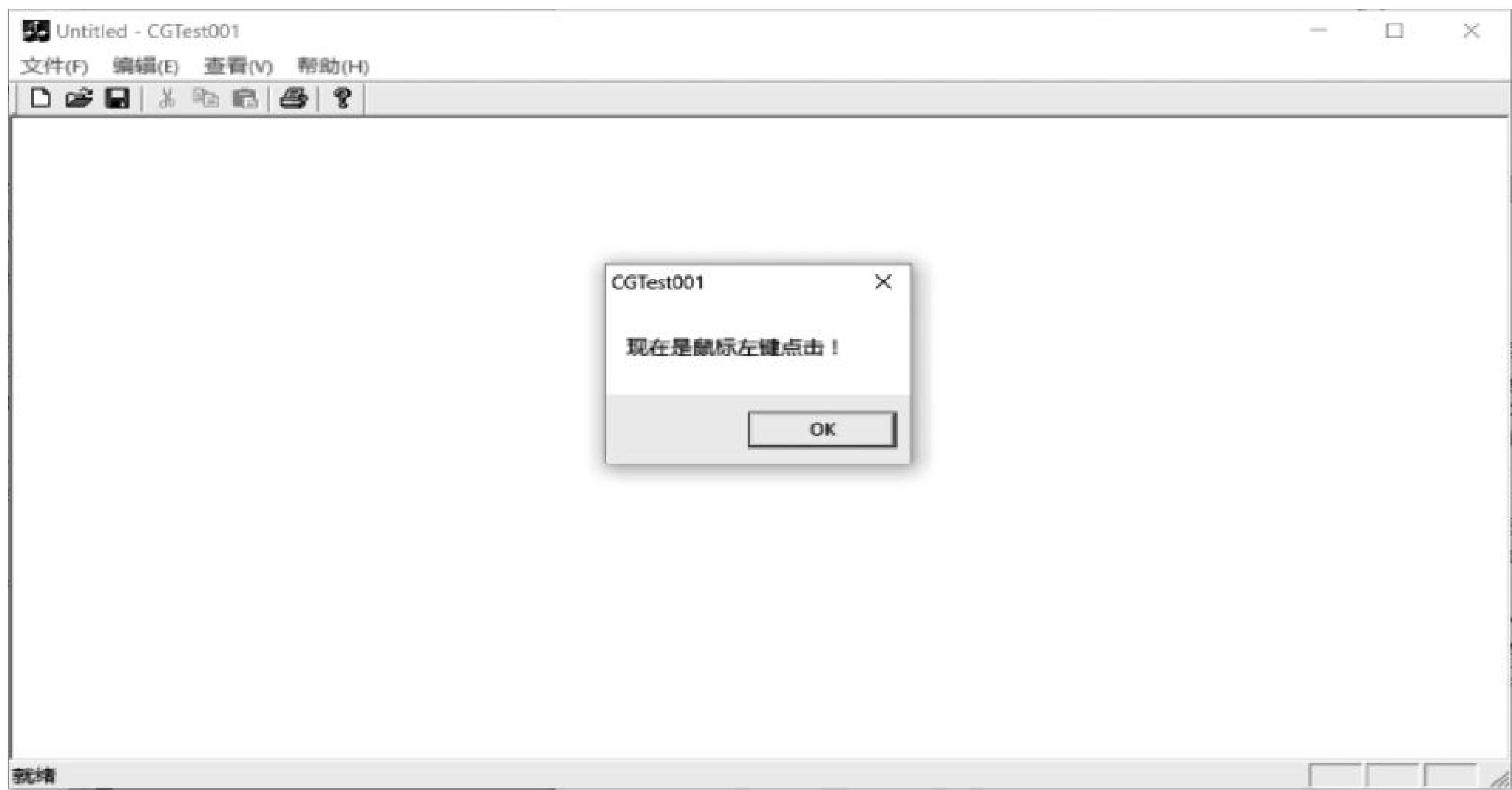


图 2.1-10　消息函数操作

应的资源属性窗口中设置相应的属性，如图 2.1-11 所示。和鼠标单击消息函数设置方法类似，单击菜单或者工具条也触发相应的消息函数，一般情况下，单击菜单和工具条是为了实现一项新的功能。

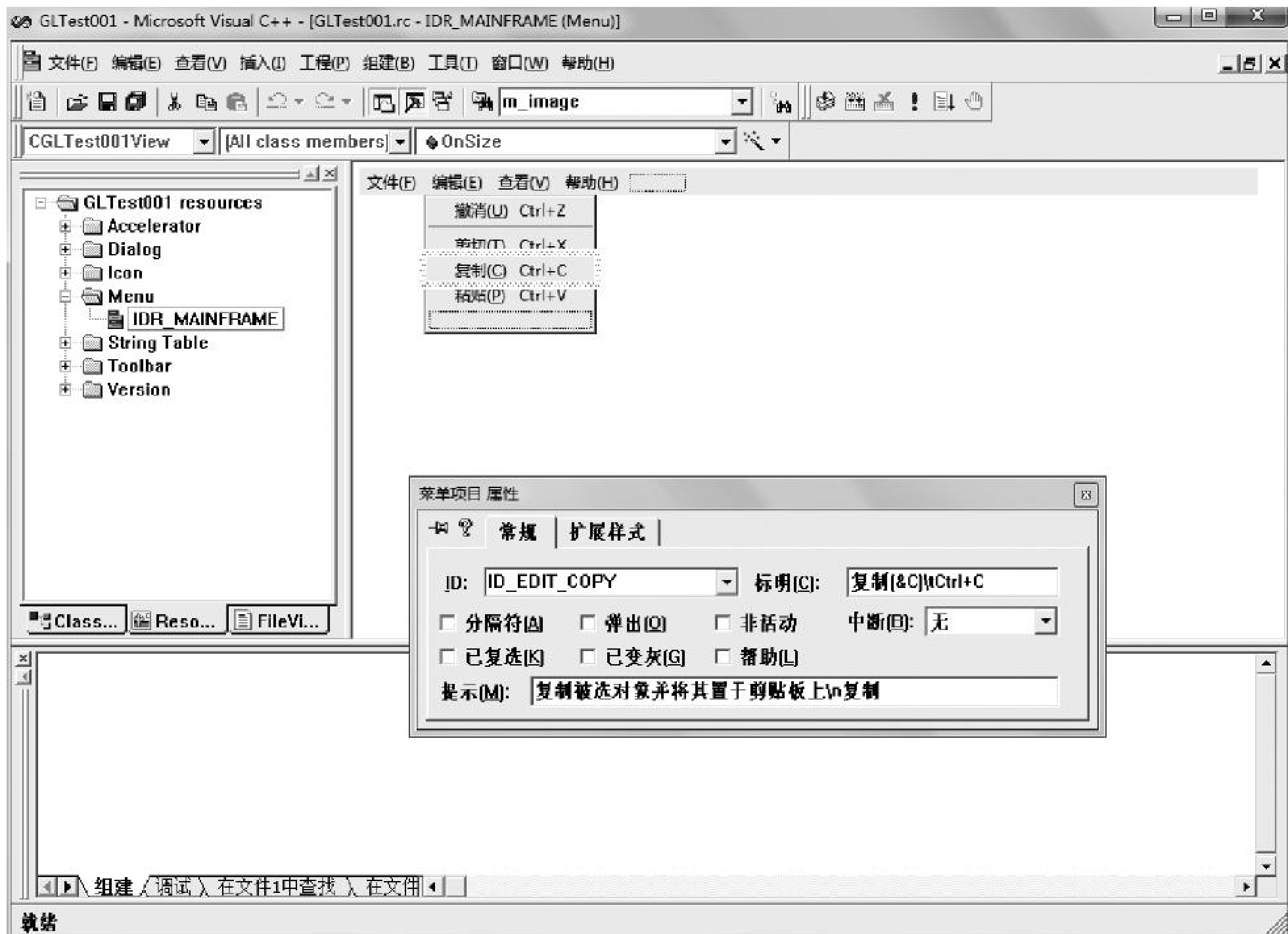


图 2.1-11　菜单属性窗口

如果要增加菜单栏或者工具条里的项目，可以双击空白菜单项或者工具条项，打开要增加的项目的属性对话框，输入唯一标识符和标题名称即可。如图 2.1-12 所示为增加一个画

线的工具条。

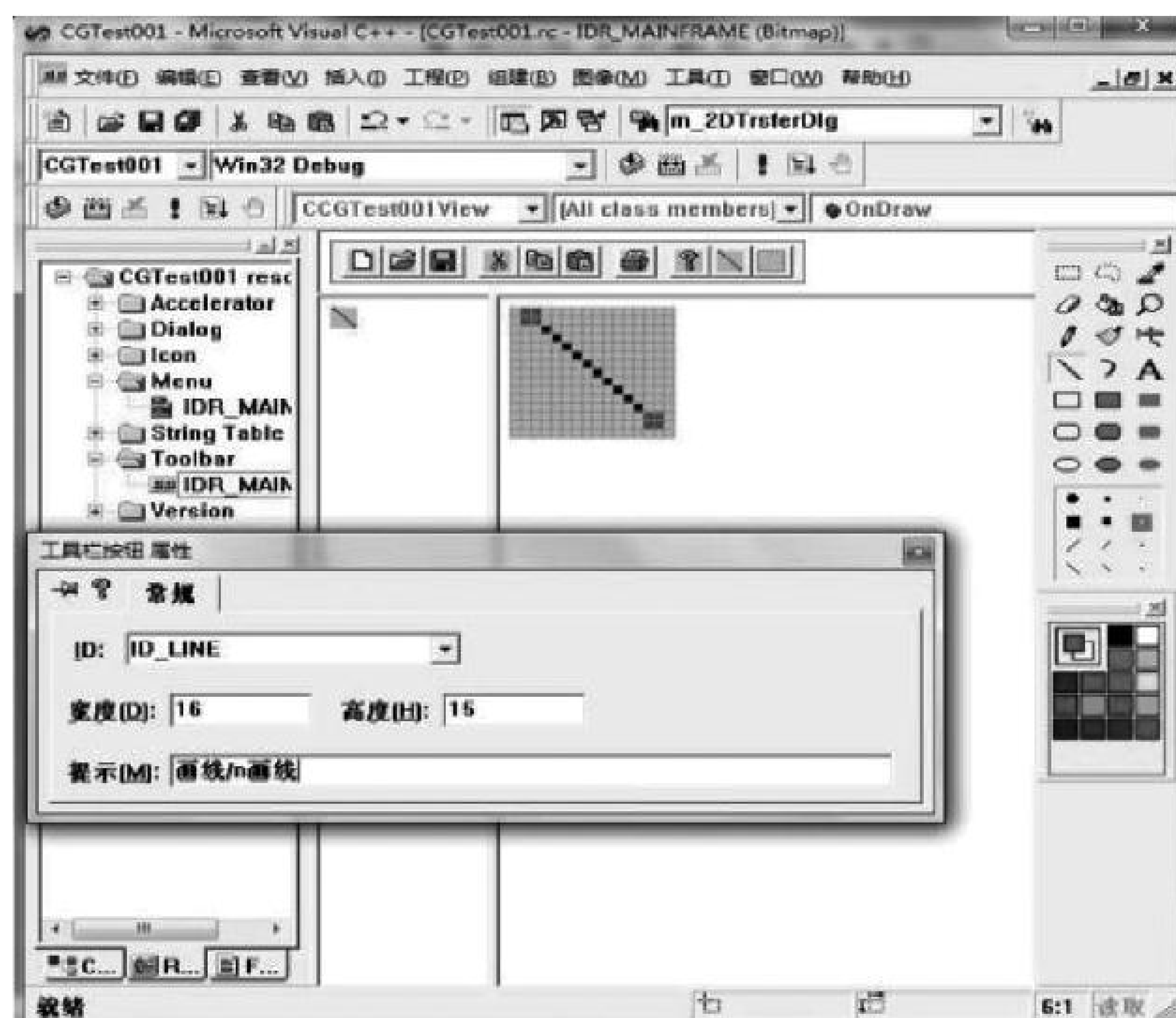


图 2.1-12　增加工具条

表 2.1-1 所示为常用的操作命令，在编译、调试或者执行应用程序时会经常用到。

表 2.1-1　常用的操作命令

工具栏图标	命令快捷方式	作 用 说 明
	Ctrl+F5 键	执行程序
	F5 键	在调试状态下执行程序
	F9 键	在光标所在行设置调试断点，再单击一次去掉断点
	Ctrl+F7 键	编译程序
	F7 键	创建程序
		停止创建程序
	F10 键	在调试状态下代码逐行执行
	Ctrl+F2 键	在光标所在行设置标签
	F2 键	锁定标签行

在 VC6.0 中，向图形设备（如显示器和打印机）的绘图和文本输入等操作是通过设备的抽象接口类 CDC 来实现的，在视图类中，有一个专门用于在设备中显示内容的函数 OnDraw(CDC ＊pDC)。例如，在绘图区域内写一行文字，直接在该函数中写代码，如图 2.1-13 所示。

使用 VC6.0 编程时，应注意以下事项：

(1) 每行代码必须以英文分号“;”结束；

(2) 代码中的变量和函数名称等的字母区分大小写；

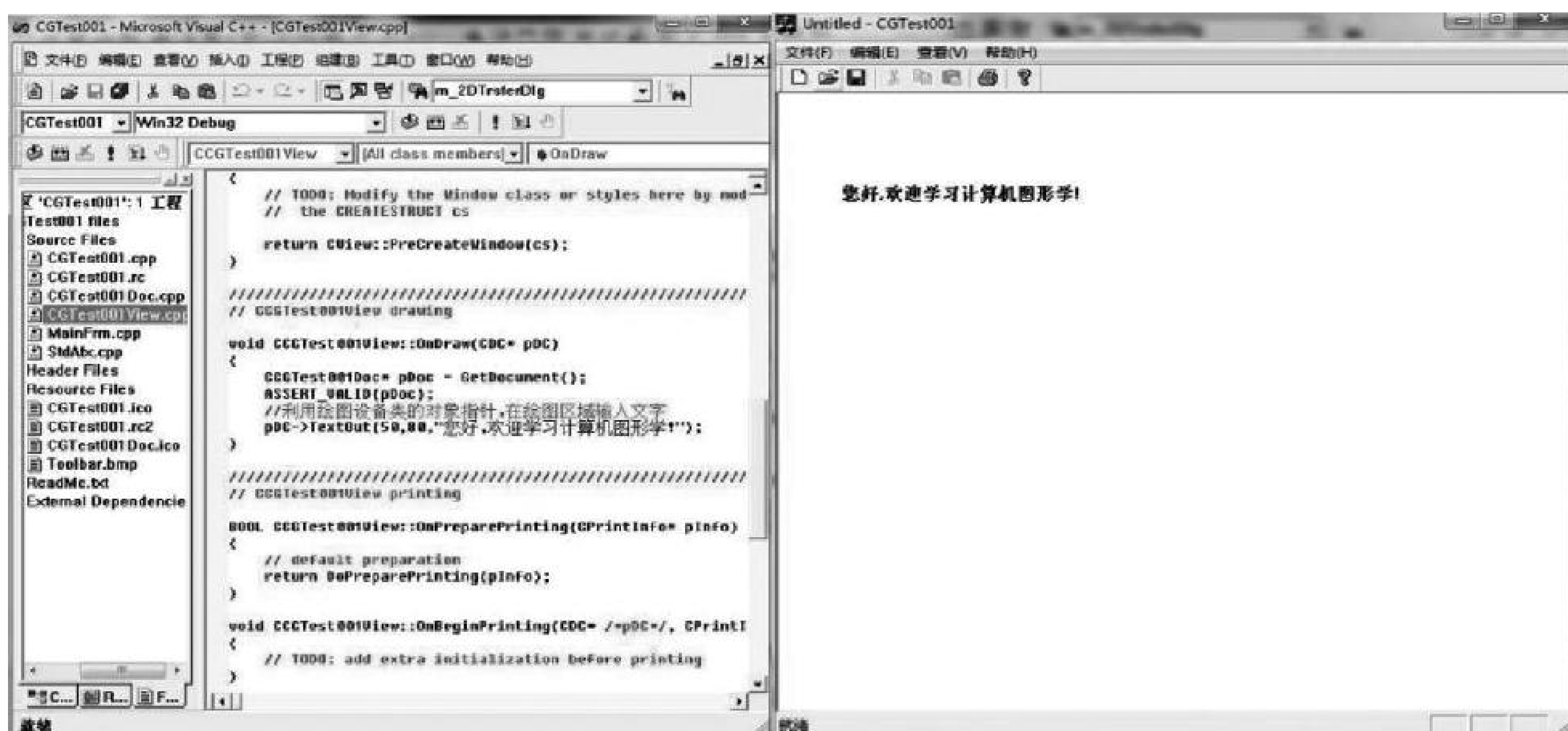


图 2.1-13　OnDraw 函数中写绘图代码

(3) 在其他函数里调用 OnDraw()时,可通过 Invalidate()函数实现;

(4) 为了交互,需要设计交互步骤,可以采用状态标识符号,如 flag=0,1,2 表示不同的状态或者操作步骤。

2.2　VC++ 基本图素的绘制方法

2.2.1　相关类及函数

在实现计算机图形学的原理和算法时,会用到 VC6.0 中的相关类、函数、数组、头文件以及模板等,本节对此进行简单介绍。

CPoint point——存放视图窗口像素点的坐标,其中 point.x 是水平方向的 x 坐标值,point.y 是垂直方向的 y 坐标值。

COLORREF crColor——像素点的颜色值类,可以直接写为 RGB(r,g,b),r、g、b 分别对应三基色中的红色值、绿色值和蓝色值,取值范围在 0~255 之间。其中 RGB(255,0,0)是红色,RGB(0,255,0)是绿色,RGB(0,0,255)是蓝色,RGB(255,255,255)是黑色。

COLORREF SetPixel(int x,int y,COLORREF crColor)——在绘图窗口的绘制点的函数,x 和 y 是点的 x 坐标和 y 坐标,crColor 是该点的颜色。例如 pDC->SetPixel(x,y,RGB(255,0,0));命令在视图窗口的(x,y)点会绘制一个红色的点。

为了在视图窗口拾取多个点,可以利用集合类 CArray 来保存这些点。CArray 是模板类,使用前需要加入模板类的头文件 afxtempl.h。例如,在视图类中使用 CArray,则在视图类的头文件 CGTest001View.h 中加入模板类的头文件 afxtempl.h:

```
#include <afxtempl.h>
```

使用时,首先在视图类头文件 CGTest001View.h 中声明点集的对象,如:

```
CArray<CPoint,CPoint> m_pt_array;
```

CArray 中常用的相关函数如下：

m_pt_array. Add(point)——在集合尾部增加一个元素(例如，增加一个点)；

m_pt_array. GetSize()——计算集合的长度；

m_pt_array. GetAt(i)——得到第 i 位元素，从第 0 位开始；

m_ptarray. RemoveAt(i)——删除集合中某个索引位置的元素；

m_ptarray. RemoveAll()——将集合内的所有元素都清空；

m_pt_array. Append(m_array)——复制另外一个集合的元素到当前的集合中。

在程序中使用数学运算时，需要加入支持数学计算函数的头文件：

```
#include <math.h>
```

2.2.2 基本像素点的交互式绘制方法

为了达到方便的交互效果，可以利用鼠标在视图窗口拾取像素点，然后记录和实时显示这些像素点。例如，每单击一次，就将该点记录在点的集合里，记录点的代码在左键单击函数 OnLButtonUp()中实现，并调用 OnDraw()函数，将点集中的点显示出来。OnLButtonUp()和 OnDraw()中的代码分别如下：

```
void CCGTest001View::OnLButtonUp(UINT nFlags, CPoint point) {
    this->m_pt_array.Add(point);            //将点加入点的集合,this 指视图窗口类本身
    Invalidate();                           //调用 OnDraw()函数
    CView::OnLButtonUp(nFlags, point);
}
void CCGTest001View::OnDraw(CDC * pDC){
    CCGTest001Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CPoint pt;
    CString str;
    for(int i = 0;i < this->m_pt_array.GetSize();i++){
        pt = this->m_pt_array.GetAt(i);     //得到第 i 个点
        //显示点,因为单个像素点显示非常不明显,故把它附近的点也绘制出来
        pDC->SetPixel(pt.x,pt.y,RGB(255,0,0));
        pDC->SetPixel(pt.x,pt.y-1,RGB(255,0,0));
        pDC->SetPixel(pt.x,pt.y+1,RGB(255,0,0));
        pDC->SetPixel(pt.x-1,pt.y,RGB(255,0,0));
        pDC->SetPixel(pt.x+1,pt.y,RGB(255,0,0));
        //文字标注点的坐标值
        str.Format("(%d,%d)",pt.x,pt.y);
        pDC->TextOut(pt.x+5,pt.y+5,str);   }
}
```

执行程序，在视图窗口每单击一次，就会以红色显示鼠标所在点，并显示该点的坐标值，如图 2.2-1 所示。

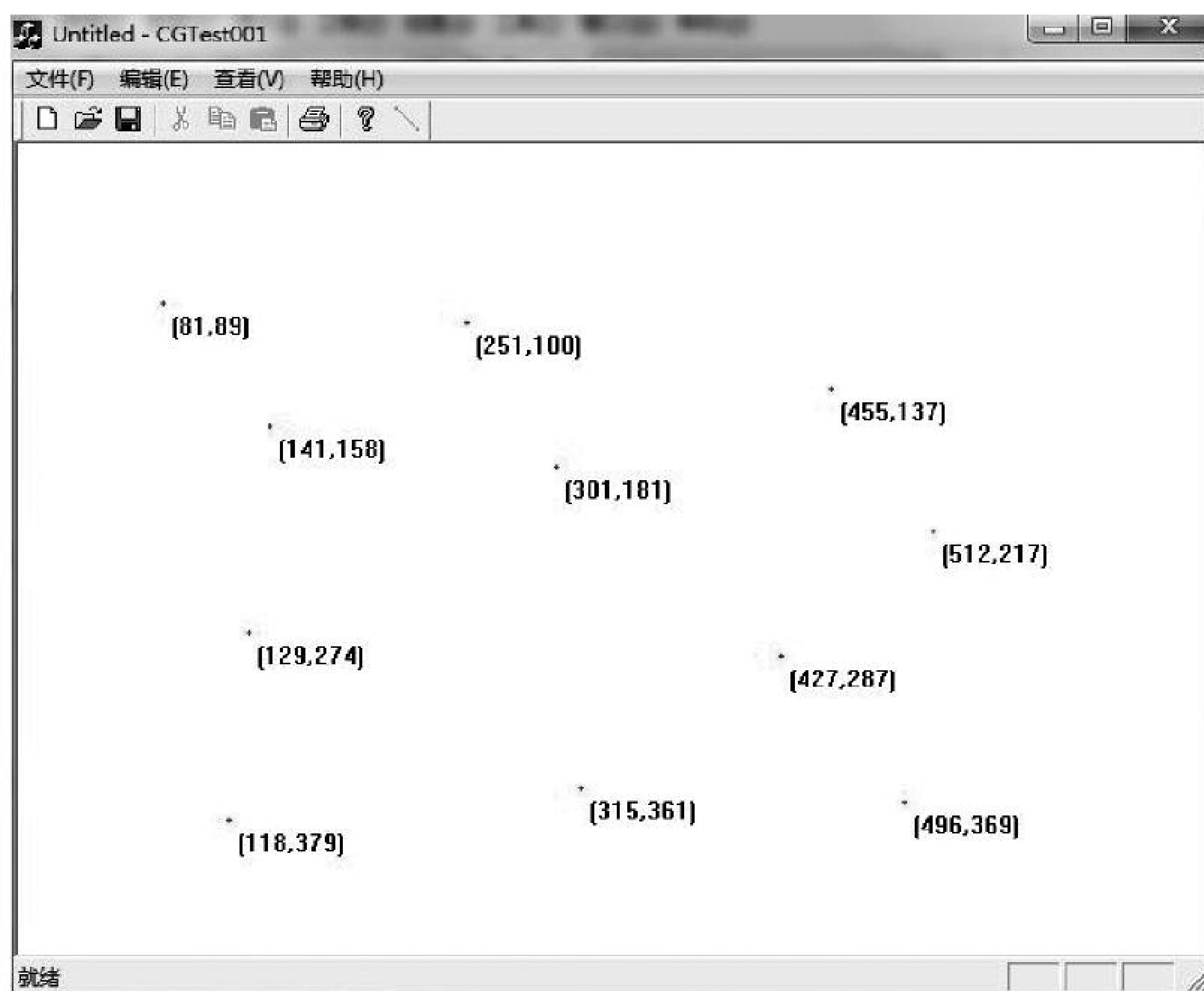


图 2.2-1　交互式绘制像素点

2.2.3　非模式对话框交互式实现方法及颜色对话框的使用

在进行计算机图形原理和算法编程时，可以采用非模式对话框实现图形实时交互。非模式对话框是指在对话框打开的同时，其他窗口还能够继续操作。非模式对话框的创建方法如下。

首先，在菜单栏中选择“插入”→“资源”命令，选择“资源类型”对话框，在新建的对话框上单击鼠标右键，选择“属性”，打开对话框属性设置窗口，在“常规”选项卡中可以修改对话框的 ID，例如 IDD_MDLG，在“更多样式”选项卡中，选中“可见”复选框，如图 2.2-2 所示。

在对话框上双击，或者右击，从弹出的快捷菜单中选择“建立类向导”命令，为创建的对话框生成类和类文件，例如，类名为 CCMDlg，则生成的对应类头文件和源文件分别为 CMDlg.h 和 CMDlg.cpp。

在 CMDlg.h 头文件的对话框类中手动添加对话框基类的 Create()函数：

```
BOOL Create();
```

在 CMDlg.cpp 执行文件中添加该函数的实现代码：

```
BOOL CCMDlg::Create(){
    return CDialog::Create(CCMDlg::IDD);
}
```

当在视图类中使用该对话框时，首先在视图类定义的代码前加入对话框的头文件：

```
#include "CMDlg.h"
```



图 2.2-2 对话框属性设置

并在类中定义对话框的指针变量：

```
CCMDlg * m_MDlg;
```

然后，在视图类 CGTest001View.cpp 的类构造函数中，建立该对话框对象指针：

```
CCGTest001View::CCGTest001View(){
    //声明创建对话框
    this->m_MDlg = new CCMDlg();
}
```

并在析构函数中，删除建立的对话框对象指针：

```
CCGTest001View::~CCGTest001View(){
    if(this->m_MDlg!= NULL){
        delete this->m_MDlg;}
}
```

可以通过菜单项或者工具栏打开对话框。例如，利用工具栏打开对话框，在视图类里，建立工具栏的消息映射函数(方法见上节)，在工具栏的消息映射函数里判断对话框是否创建，如没有创建，则创建；如已创建，则显示。代码如下：

```
void CCGTest001View::OnLine() {
    if(this->m_MDlg->GetSafeHwnd() == NULL){
```

```
        this->m_MDlg->Create();}
    else
        this->m_MDlg->ShowWindow(TRUE);
}
```

执行程序，打开软件，单击工具栏上的图标，即可打开对话框，如图 2.2-3 所示，这时仍可在视图窗口操作。

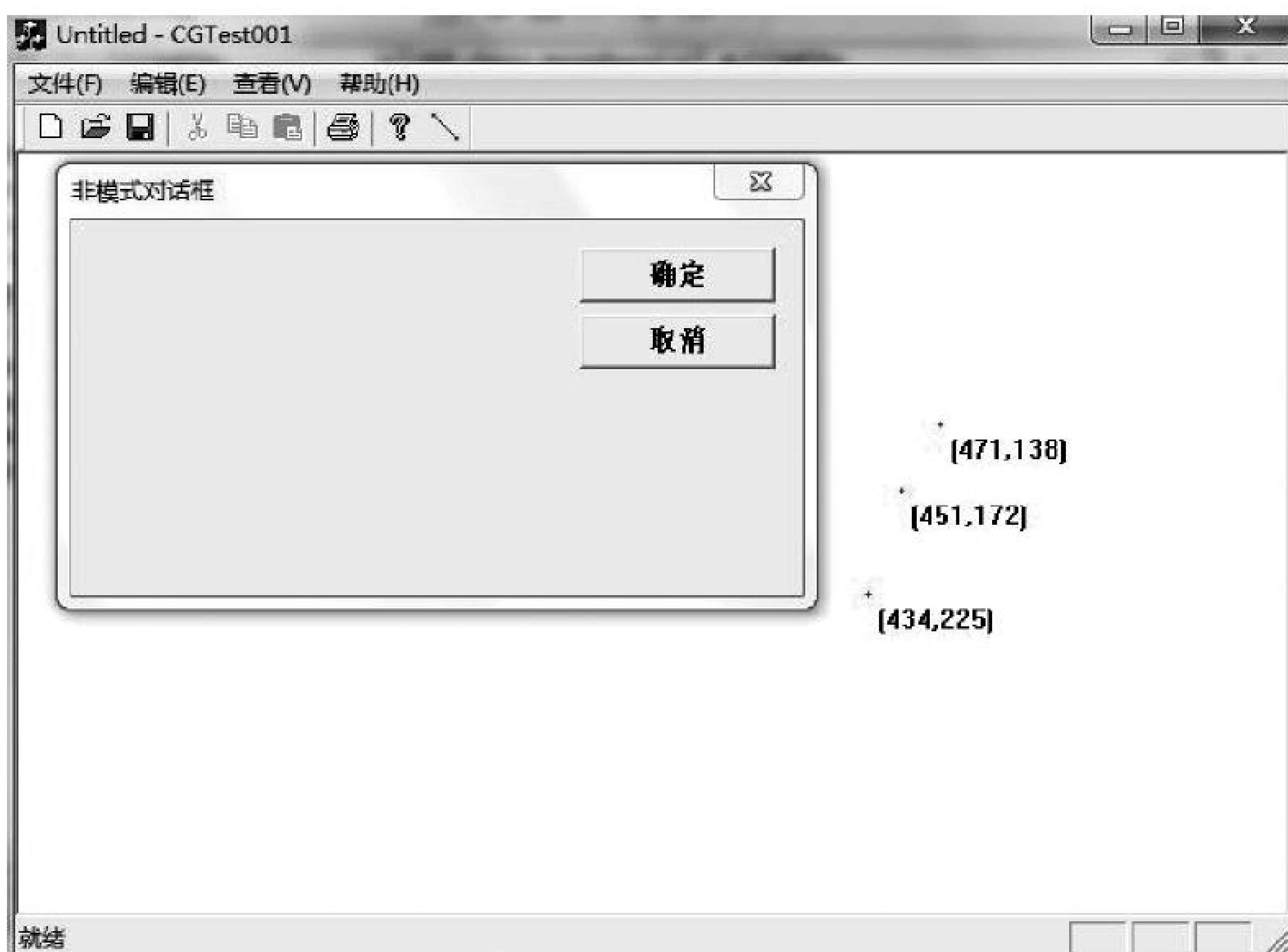


图 2.2-3　创建非模式对话框

视图窗口的数据可以传入到对话框中。例如，将视图窗口上点的坐标传到对话框中，首先在对话框中放入两个从控件中拖入的编辑框来显示坐标值，ID 分别为 IDC-EDITX 和 IDC-EDITY，通过在菜单栏中选择“建立类向导”→在 Member Variables 选项卡中选择对话框类，分别选中两个编辑框 ID，单击“Add Variable...”按钮，如图 2.2-4 所示，为两个编辑框建立变量名 m_X 和 m_Y，变量类型都是 int 整型。

在视图窗口单击时，如果对话框是创建好的，则把当前点的坐标值赋给对话框里的编辑框，并实时显示出来。代码如下：

```
void CCGTest001View::OnLButtonUp(UINT nFlags, CPoint point) {
    this->m_pt_array.Add(point);              //this 指的是视图窗口这个类本身
    if(this->m_MDlg->GetSafeHwnd()!= NULL){
        this->m_MDlg->m_X = point.x;
        this->m_MDlg->m_Y = point.y;
        this->m_MDlg->UpdateData(FALSE);
    }
    Invalidate();                             //调用 OnDraw()函数
    CView::OnLButtonUp(nFlags, point);
}
```

执行程序，当在视图窗口单击时，当前点的坐标值会实时显示在对话框中，如图 2.2-5 所示。

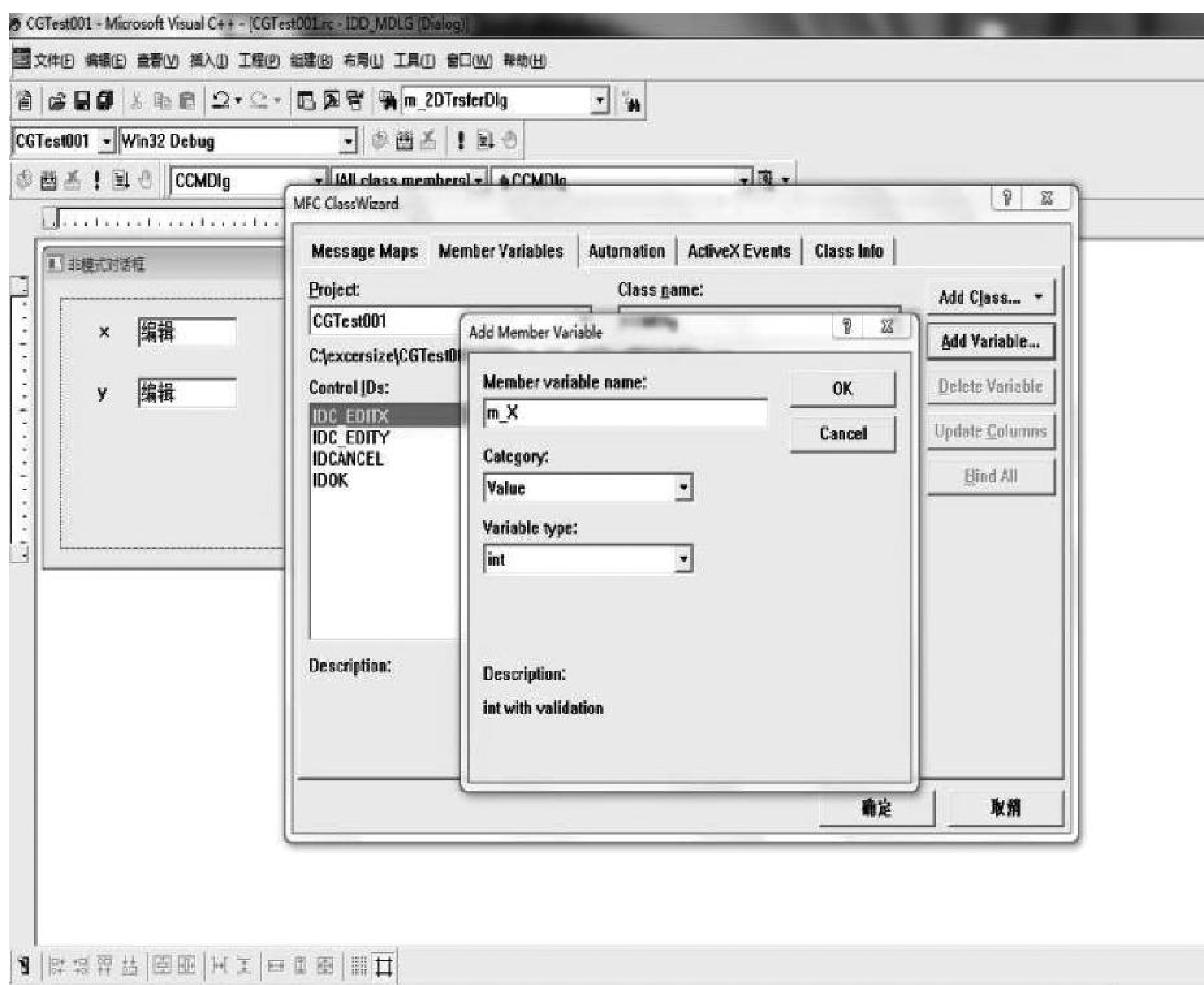


图 2.2-4 建立变量



图 2.2-5 对话框实时显示视图窗口的数据

如果要把对话框里的数据传给视图窗口，则在对话框类中加入视图窗口的类变量，通过视图窗口的类变量把对话框的数据赋给视图窗口。首先，在对话框类头文件 CMDlg.h 中添加视图窗口类的引用：

```
class CCGTest001View;
```

并在 CCMDlg 类中增加视图窗口类指针变量：

```
CCGTest001View *m_pView;
```

在源文件 CMDlg.cpp 增加对视图窗口类文件的引用：

```
#include "CGTest001View.h"
```

在视图窗口类 CCGTest001View 的构造函数中，在声明创建对话框后，还需要将视图窗口类的指针赋给对话框类中的视图窗口类变量：

```
CCGTest001View::CCGTest001View(){
    //声明创建对话框
    this->m_MDlg = new CCMDlg();
    this->m_MDlg->m_pView = this;
}
```

进行上述设置后编译程序，如果出现关于 GetDocument 的相关错误，还需要在 CGTest001View.h 中增加对 CCGTest001Doc 文档类的引用：

```
class CCGTest001Doc;
```

这样，对话框中的视图窗口对象就和当前的视图窗口建立了联系。在对话框中输入点的坐标值，通过"确定"按钮的消息函数，就将该坐标值传给视图窗口的点集合中，并显示出来：

```
void CCMDlg::OnOK() {
    UpdateData(TRUE);
    CPoint pt;
    pt.x = this->m_X;
    pt.y = this->m_Y;
    this->m_pView->m_pt_array.Add(pt);
    this->m_pView->Invalidate();
    //CDialog::OnOK();删除该行代码
}
```

程序执行效果如图 2.2-6 所示。单击"确定"按钮就可以把对话框中的坐标值加入点集合中，并在视图窗口显示出来。使用上述方法就可以实现对话框和视图窗口的双向交互。

在应用程序中绘制图形时，会为图形设置各种颜色，可以直接调用类似 RGB(0～255，0～255，0～255)这个结构来设置颜色，也可以通过 VC6.0 提供的颜色对话框类 CColorDialog 获取颜色。CColorDialog 对话框对象通过 DoModal()函数直接打开，然后通过调用 CColorDialog 的一个结构体 m_cc 的成员 rgbResult 来获取选定的颜色值，再将这个值保存在设定的颜色变量中，就可以利用这个变量来设置颜色。代码如下：

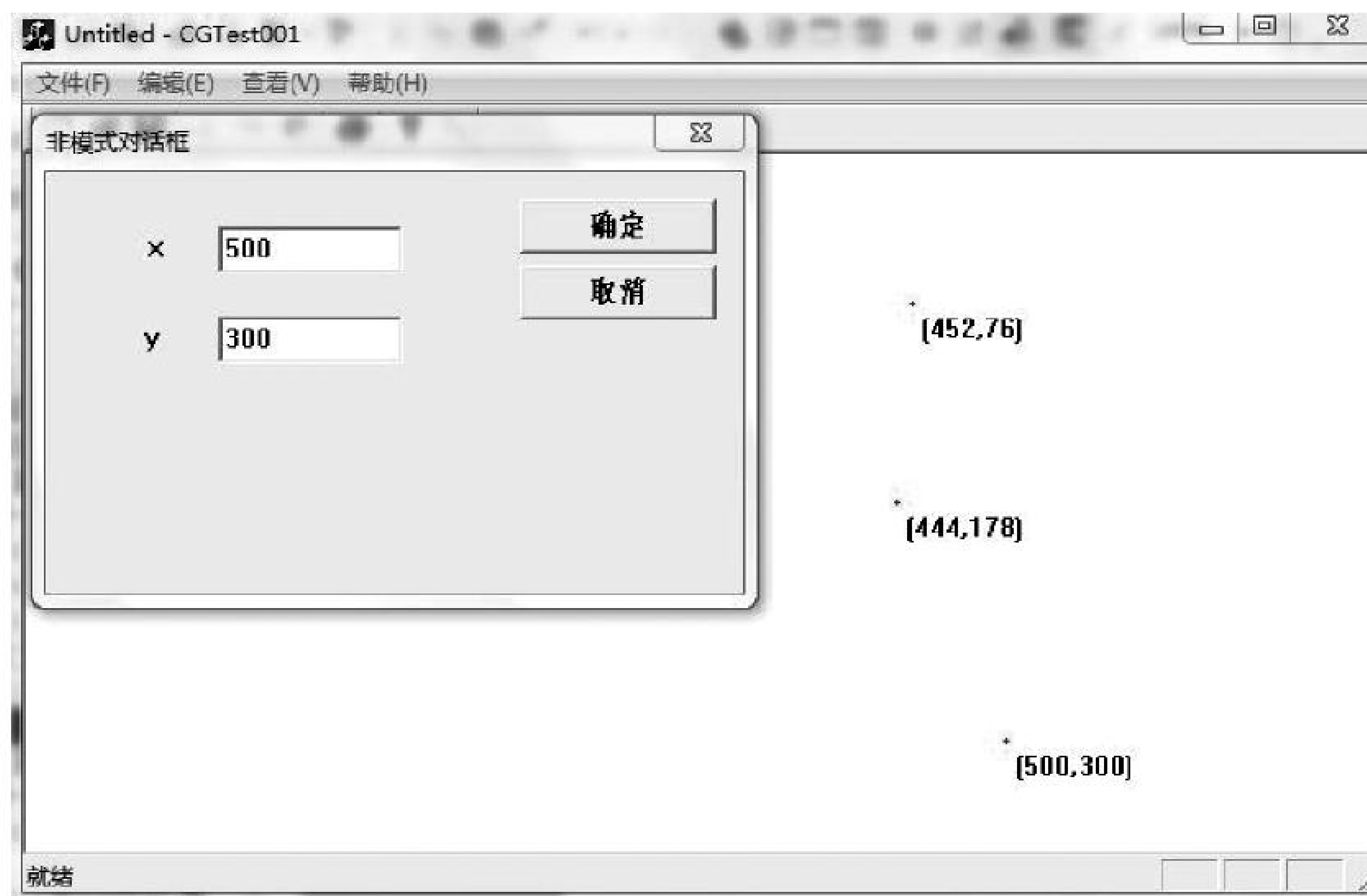


图 2.2-6 对话框中数据在视图窗口显示

```
CColorDialog dlg;
dlg.m_cc.Flags| = CC_RGBINIT|CC_FULLOPEN;
if(IDOK == dlg.DoModal()){
    COLORREF m_clr = dlg.m_cc.rgbResult;  //将 dlg.m_cc.rgbResult 获取到的颜色对话框中的颜色保存到变量 m_clr 中
    int iR = GetRValue(m_clr);            //颜色中红色的强度值
    int iG = GetGValue(m_clr);            //颜色中绿色的强度值
    int iB = GetBValue(m_clr);            //颜色中蓝色的强度值
}
```

第 3 章 基本图形的生成

计算机图形学首先要解决的问题是在图形输出设备(例如显示器)上生成和显示基本的图形元素,例如点、直线、圆弧、椭圆以及其他二维图形等。但是,由于图形本身是几何连续的形状,而显示设备却是由离散的像素点组成的像素点阵,例如,显示器的分辨率 1024×768 是指显示器屏幕在水平方向每一行有 1024 个像素点,在垂直方向每一列有 768 个像素点,由此组成了屏幕的像素点阵,那么这些离散的像素点是不能完全真实地表示连续图形的。为了实现在屏幕上显示图形,可以通过寻找屏幕上的一组像素点集,并将该组像素点集用指定的颜色显示,以此来最佳逼近图形的形状,这种图形的显示方法称为图形的扫描转换,也称为图形的光栅化。

在第 2 章图形开发工具的相关内容中已经实现了基本图形之一——点的生成和显示方法,本章将实现其他基本图形的扫描转换。

3.1 直线的扫描转换

3.1.1 直线扫描转换原理

直线的扫描转换是指在图形输出设备上,按照扫描线的顺序,确定一组最佳逼近于直线的像素点并对像素点进行写操作。如图 3.1-1 所示为用一组像素点来代表直线。

因此,直线生成要解决的具体问题是:已知直线的两个端点 $P_0(x_0,y_0)$ 和 $P_1(x_1,y_1)$,则要求在图形输出设备上,从起点到终点通过逐次循环迭代,找到最接近直线的像素点。所以需要建立循环迭代的增量方程:$x_{i+1}=x_i+\Delta x$,$y_{i+1}=y_i+\Delta y$。

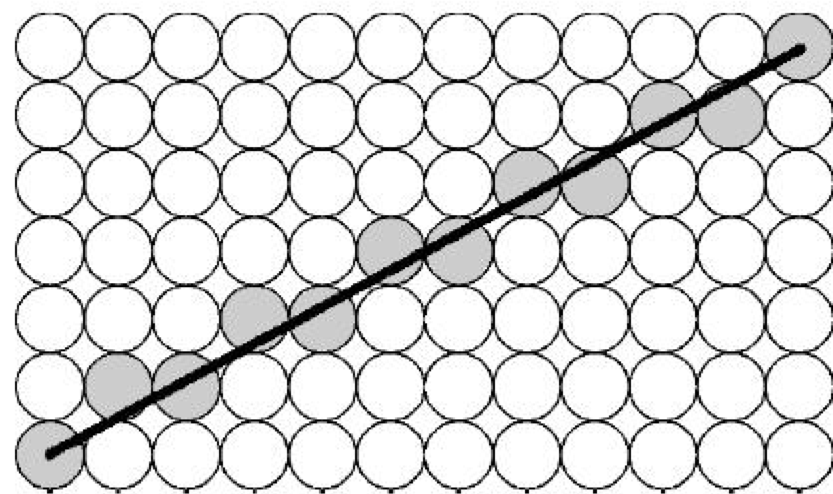


图 3.1-1　直线的扫描转换

直线的扫描转换算法的目的就是建立合适的增量方程算法。由于图形可能包含很多条直线,所以,要求直线扫描转换算法的绘制速度要尽可能得快,尽量不要出现浮点数占用大的内存和避免乘、除、开方及角度等复杂运算。

比较常用的直线扫描转换算法有三种:数值微分法(DDA);中点画线算法;Bresenham 算法。

3.1.2 数值微分法

数值微分法(digital differential analyzer,DDA)是直接利用直线斜率的增量方程来计算直线上下一个迭代像素点的方法。

直线的斜率微分方程为

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0} = k$$

数值微分法的迭代公式如下：

$$x_{i+1} = x_i + \varepsilon \Delta x$$
$$y_{i+1} = y_i + \varepsilon \Delta y$$

其中

$$\varepsilon = 1/\max(|\Delta x|, |\Delta y|)$$

当 $\max(|\Delta x|, |\Delta y|) = |\Delta x|$ 时,则

$$x_{i+1} = x_i + \varepsilon \Delta x = x_i + \frac{1}{|\Delta x|}\Delta x = x_i \pm 1$$
$$y_{i+1} = y_i + \varepsilon \Delta y = y_i + \frac{1}{|\Delta x|}\Delta y = y_i \pm k$$

当 $\max(|\Delta x|, |\Delta y|) = |\Delta y|$ 时,则

$$x_{i+1} = x_i + \varepsilon \Delta x = x_i + \frac{1}{|\Delta y|}\Delta x = x_i \pm \frac{1}{k}$$
$$y_{i+1} = y_i + \varepsilon \Delta y = y_i + \frac{1}{|\Delta y|}\Delta y = y_i \pm 1$$

数值微分法的算法示意图如图 3.1-2 所示。当直线的斜率 k 绝对值小于 1 时,迭代在 x 方向步进,x 向步长为 1(个像素),y 方向的步进为 $y_{i+1}=y_i \pm k$,y 方向对应的像素点坐标值为 round (y_{i+1}),即 y 方向取最接近 y_{i+1} 的整数值,故获得的下一个像素点为 $(x_{i+1}$,round$(y_{i+1}))$;当直线的斜率 k 绝对值大于 1 时,迭代在 y 方向步进,y 向步长为 1(个像素),x 方向的步进为 $x_{i+1}=x_i \pm \frac{1}{k}$,$x$ 方向对应的像素点坐标值为 round(x_{i+1}),即 x 方向取最接近 x_{i+1} 的整数值,下一个最佳像素逼近点为(round(x_{i+1}),y_{i+1})。

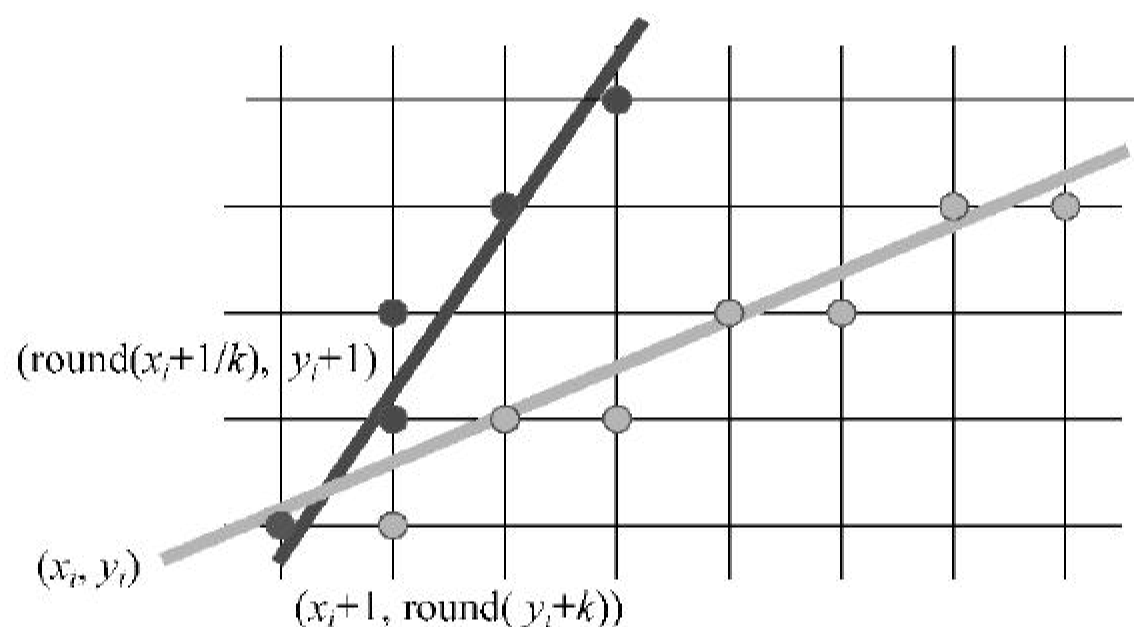


图 3.1-2 数值微分法示意图

在编写直线生成算法的程序时，除了实现一般位置的直线算法外，还应该考虑特殊位置直线的生成，例如，y 方向的竖直线和 x 方向的水平线，这样才能获得稳定的算法。下面是考虑各种情况的直线数值微分法算法函数参考代码：

```
/***********************************************************************
DDA_Line: 直线的数值微分算法函数
pDC: 显示器指针; startPoint: 起点; endPoint: 终点; crColor: 线的颜色
***********************************************************************/
void DDA_Line(CDC *&pDC,CPoint &startPoint,CPoint &endPoint,COLORREF crColor){
    if(endPoint.x!= startPoint.x&&endPoint.y!= startPoint.y){  //非特殊位置直线,一般直线
        double x,y;
        double k;                          //斜率
        k = ((float)(endPoint.y - startPoint.y))/((float)(endPoint.x - startPoint.x));
        x = (double)startPoint.x;
        y = (double)startPoint.y;
        pDC->SetPixel((int)x,(int)y,crColor);
        if(abs(k)<= 1.0){
           for(int i = 0;i < abs(endPoint.x - startPoint.x);i++){  //x 增量
               if(endPoint.x > startPoint.x){
                    x += 1;
                    y += k; }
                else  {
                    x -= 1;
                    y -= k; }
                pDC->SetPixel((int)x,(int)(y + 0.5),crColor);
            }
        }
        else if(abs(k)> 1.0)  {
            for(int i = 0;i < abs(endPoint.y - startPoint.y);i++){  //y 增量
                if(endPoint.y > startPoint.y){
                    y += 1;
                    x += 1.0/k;
                }
                else  {
                    y -= 1;
                    x -= 1.0/k;
                }
                pDC->SetPixel((int)(x + 0.5),(int)(y),crColor);
            }
        }
    }
    else if(startPoint.x == endPoint.x){     //垂直画线
        if(startPoint.y < endPoint.y){
            for(int i = startPoint.y;i <= endPoint.y;i++){
                pDC->SetPixel(startPoint.x,i,crColor);
            }
        }
        else if(startPoint.y > endPoint.y){
```

```
                for(int i = startPoint.y;i >= endPoint.y;i -- )  {
                    pDC -> SetPixel(startPoint.x,i,crColor);
                }
            }
        }
        else if(startPoint.y == endPoint.y){      //画水平线
            if(startPoint.x < endPoint.x){
                for(int i = startPoint.x;i < endPoint.x;i++)  {
                    pDC -> SetPixel(i,startPoint.y,crColor);
                }
            }
            else if(startPoint.x > endPoint.x){
                for(int i = startPoint.x;i >= endPoint.x;i -- )  {
                    pDC -> SetPixel(i,startPoint.y,crColor);
                }
            }
        }
}
```

将直线生成函数保存为单独的文件，如文件名 BasicGraph.h，并在应用程序中直接引用：

```
#include "BasicGraph.h"
```

在应用程序中生成直线时，可以采用第 2 章的有关代码建立应用程序 CGTest002，并在视图窗口拾取多个点，利用上述的中点画线函数顺序连接前后点生成直线。实现方法如下。

在视图窗口类的源文件 CGTest002View.cpp 中加入对 BasicGraph1.h 的引用，在 OnDraw()函数中就可以直接调用上述的直线数值微分法函数：

```
void CCGTest002View::OnDraw(CDC * pDC){
    CCGTest002Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CPoint startPoint, endPoint;
    if(this -> m_pt_array.GetSize()> 1){
        //两点以上开始画线
        startPoint = this -> m_pt_array.GetAt(0);  //得到第 1 个点
        for(int i = 1;i < this -> m_pt_array.GetSize();i++){  //循环画线
            endPoint = this -> m_pt_array.GetAt(i);
            DDA_Line(pDC,startPoint,endPoint,RGB(255,0,0));
            startPoint = endPoint;
        }
    }
}
```

执行程序，在视图窗口拾取多个点，生成直线如图 3.1-3 所示。

数值微分法的特点是增量算法，直观、易实现。但是算法中有除法运算和浮点数，不利于用硬件实现；当图形中有大量的直线时，利用数值微分法会占用较多的内存，对运算速度会有一些影响。

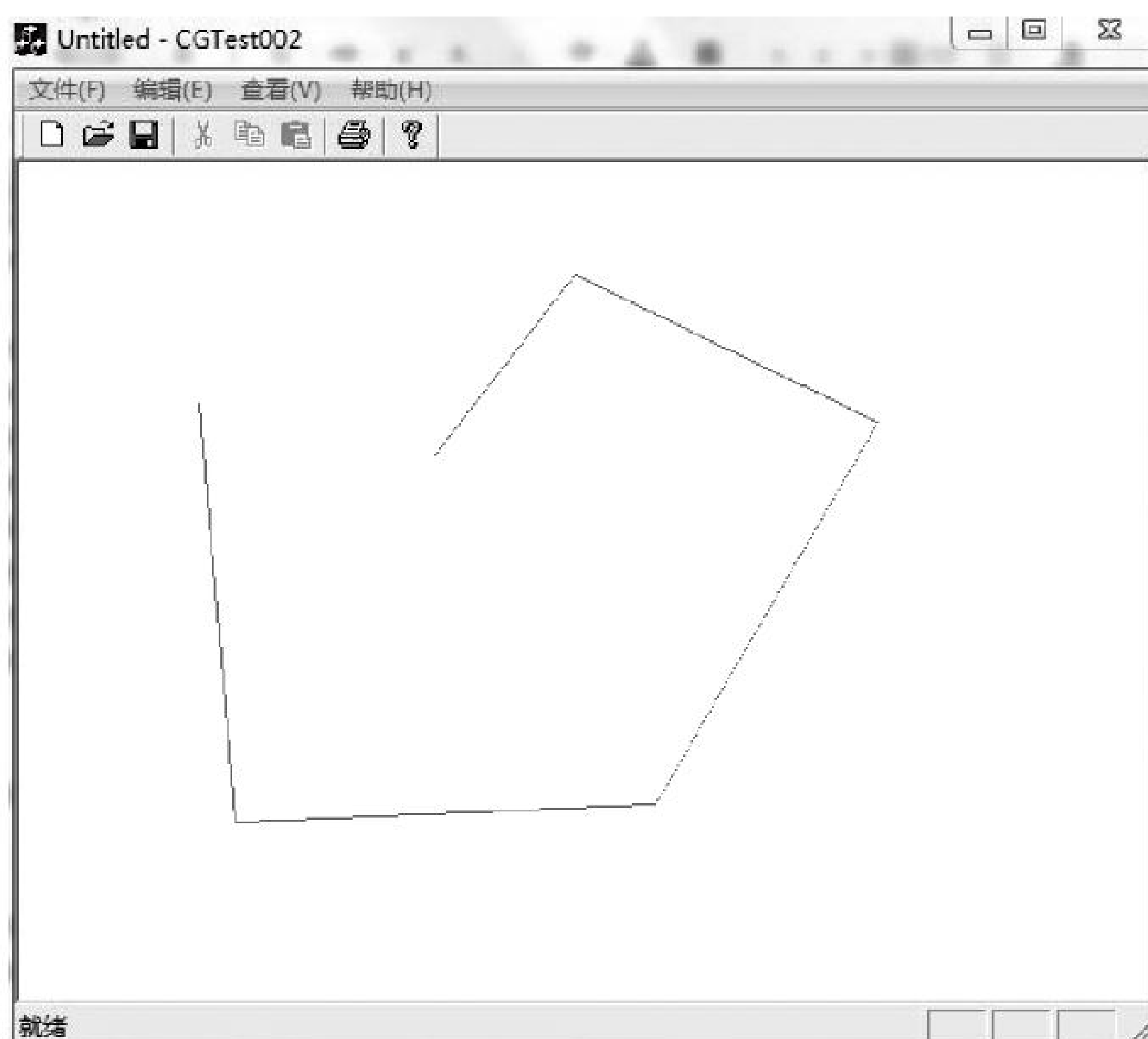


图 3.1-3　数值微分法画线

3.1.3　中点画线算法

直线的中点画线算法是一种很巧妙的方法，在算法推导过程中使用了中点的概念，但是进行循环迭代时，却是根据正负值判断，和中点并没有关系。

中点画线法是借助直线的隐式方程的相关理论来实现的，假设直线的两个端点为 $P_0(x_0,y_0)$ 和 $P_1(x_1,y_1)$，直线段隐式方程可表示为

$$F(x,y)=ax+by+c=0$$

式中

$$a=y_0-y_1,\quad b=x_1-x_0,\quad c=x_0y_1-x_1y_0$$

直线方程将空间分成三个区域，其中直线上方的点，将其 x、y 值代入方程中，$F(x,y)>0$，直线下方的点 $F(x,y)<0$，直线上的点 $F(x,y)=0$，如图 3.1-4 所示。

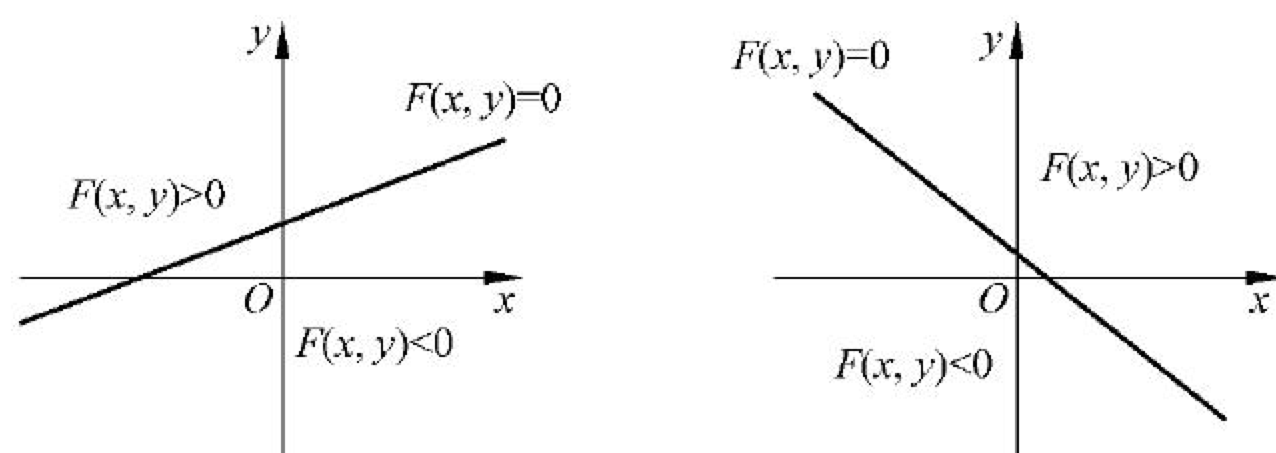


图 3.1-4　隐式方程对空间区域的划分

假定直线的斜率 $0<k<1$，则迭代在 x 方向步进，为了判断下一个最佳像素逼近点的位置，首先构造点 $(x_i+1,y_i+0.5)$ 的判别式：$d=F(x_i+1,y_i+0.5)$。

如图 3.1-5(a)、(b)所示，当判别式 $d \geqslant 0$ 时，直线在点 $(x_i+1, y_i+0.5)$ 的上方，这说明直线更接近该点下方的像素点，则取下方的像素点 (x_i+1, y_i)；当判别式 $d<0$ 时，直线在点 $(x_i+1, y_i+0.5)$ 的下方，这说明直线更接近该点上方的像素点，则取上方的像素点 (x_i+1, y_i+1)。

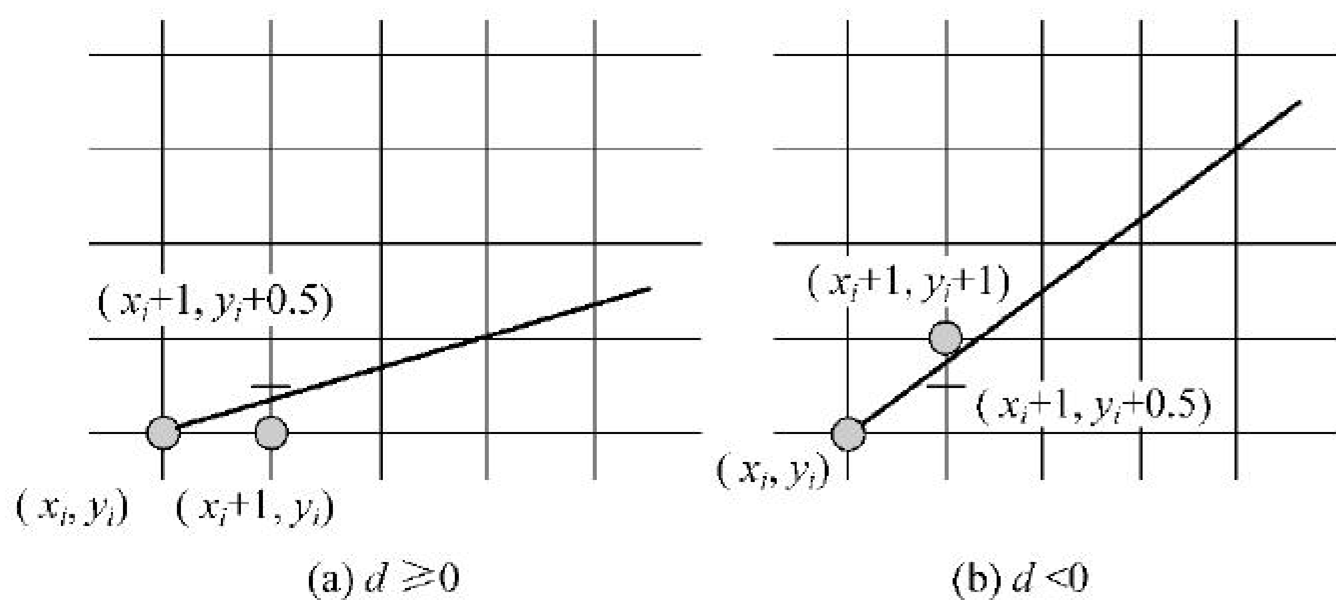


图 3.1-5 判别式 $d=F(x_i+1, y_i+0.5)$ 的情况

然后，再构造下一个点的判别式，这时需要根据上面两种情况分别构造。

(1) 当 $d \geqslant 0$ 时，则下一个构造点为 $(x_i+2, y_i+0.5)$，如图 3.1-6(a)所示，判别式为

$$d_1 = F(x_i+2, y_i+0.5) = a(x_i+1+1)+b(y_i+0.5)+c = d+a$$

此时，d_1 的增量是 a，是个整数。

(2) 当 $d<0$ 时，则下一个构造点为 $(x_i+2, y_i+1.5)$，如图 3.1-6(b)所示，判别式为

$$d_1 = F(x_i+2, y_i+1.5) = a(x_i+1+1)+b(y_i+1.5)+c = d+a+b$$

此时，d_1 的增量是 $a+b$，也是个整数。

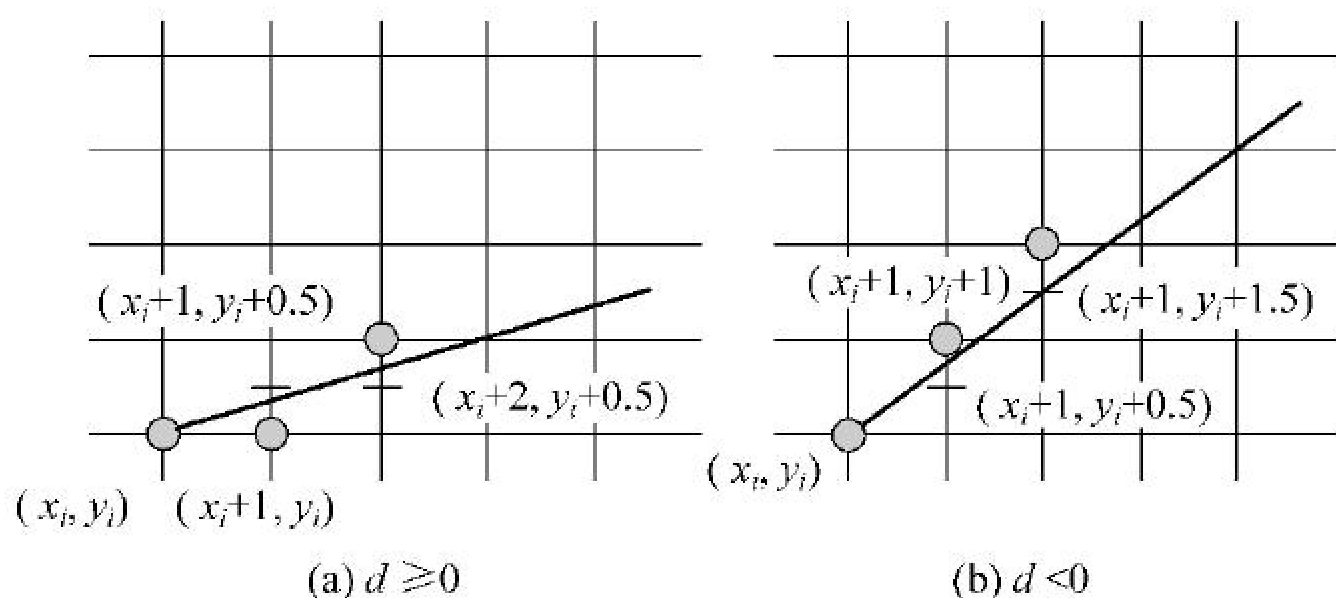


图 3.1-6 判断 x_i+2 的情况

上述是迭代过程中判别式的增量，现在计算判别式的初始值：

$$d_0 = F(x_0+1, y_0+0.5) = F(x_0, y_0)+a+0.5b$$

因为点 $P_0(x_0, y_0)$ 在直线上，故 $F(x_0, y_0)=0$，那么，判别式的初始值为

$$d_0 = a+0.5b$$

判别式 d 的增量是整数，只有初始值中包含小数，那么可以用 $2d$ 代替 d，这样在整个算法中就不再出现小数，只有整数运算，即

$$d_0 = 2(a+0.5b) = 2a+b$$

当 $d_i \geqslant 0$ 时，取 (x_i+1, y_i)，则 $d_{i+1}=d_i+a \Rightarrow d_{i+1}=d_i+2a$；

当 $d_i<0$ 时，取点 (x_i+1, y_i+1)，则 $d_{i+1}=d_i+a+b \Rightarrow d_{i+1}=d_i+2(a+b)$。

例如：用中点画线法连接两点 $P_0(0,0)$ 和 $P_1(5,4)$ 的直线段，如图 3.1-7 所示。

计算过程如下：

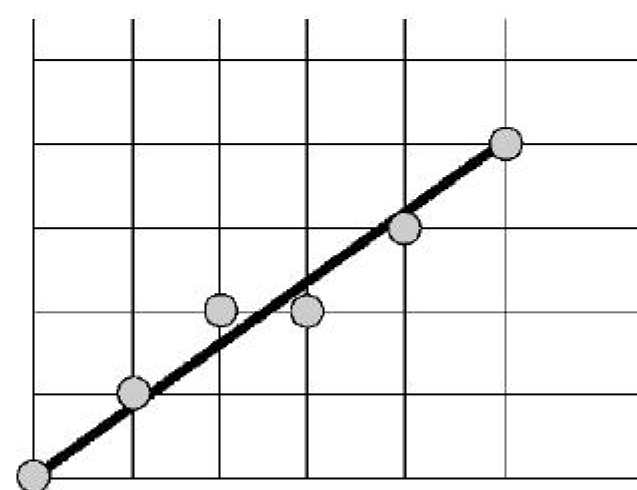


图 3.1-7 中点画线法示意

$a = y_0 - y_1 = 0 - 4 = -4$

$b = x_1 - x_0 = 5 - 0 = 5$

$d_0 = 2a + b = -3 < 0 \quad \Rightarrow (1,1)$

$2a = -8$

$2(a + b) = 2$

$d_1 = d_0 + 2(a + b) = -1 < 0 \quad \Rightarrow (2,2)$

$d_2 = d_1 + 2(a + b) = 1 > 0 \quad \Rightarrow (3,2)$

$d_3 = d_2 + 2a = -7 < 0 \quad \Rightarrow (4,3)$

$d_4 = d_3 + 2(a + b) = -7 + 2 = -5 < 0 \quad \Rightarrow (5,4)$

上述是 $0<k\leqslant 1$ 的算法，当 $k>1$ 时，迭代在 y 方向步进，需要按照上述方法重新推导判别式初始值 d_0 和迭代时的判别式值 d_1。在编程实现中点画线算法时，和数值微分法一样，需要考虑其他情况，例如，斜率 $-1\leqslant k<0$，$k\leqslant -1$，端点的起始终止顺序和直线特殊位置，这时需要进行相应处理，使算法保持稳定性。下面是一般位置直线的中点画线算法参考代码(特殊位置直线算法代码省略)：

```
/*************************************************************************
MIDPOINT_Line: 直线的中点画线算法函数
pDC: 显示器指针; startPoint: 起点; endPoint: 终点; crColor: 线的颜色
*************************************************************************/
void MIDPOINT_Line(CDC *&pDC,CPoint startPoint,CPoint endPoint,COLORREF crColor){
    if(endPoint.x!= startPoint.x&&endPoint.y!= startPoint.y){   //非特殊位置直线,一般直线
        int kFlag = 0;                      //0:斜率≤1,1:斜率>1
        int sFlag = 1;                      //斜率正负的标识
        if(startPoint.x > endPoint.x){      //首先判断两个端点,使起点 x 小于终点 x
            CPoint pt = startPoint;
            startPoint = endPoint;
            endPoint = pt;   }
        if(abs(endPoint.y - startPoint.y)> abs(endPoint.x - startPoint.x))
            kFlag = 1;                      //如果斜率大于 1
        if(startPoint.y > endPoint.y)       //如果斜率小于 0,则进行标识
            sFlag =- 1;
        int a,b,tA,tAB,d,x,y;
        if(sFlag ==- 1)
            endPoint.y = startPoint.y + (startPoint.y - endPoint.y);
        a = startPoint.y - endPoint.y;
        b = endPoint.x - startPoint.x;
        tA = 2 * a;
        tAB = 2 * (a + b);
        d = 2 * a + b;
        x = startPoint.x;
        y = startPoint.y;
        pDC - > SetPixel(x,y,crColor);
        if(kFlag == 0){                                      //斜率≤1
            for(int i = 0;i <(endPoint.x - startPoint.x);i++){
```

```
                if(d>=0)  {
                    pDC->SetPixel(x+1,y,crColor);          //取点(x+1,y)
                    x+=1;
                    d+=tA;
                }
                else {//d<0
                    pDC->SetPixel(x+1,y+sFlag,crColor);    //取点(x+1,y+1)
                    x+=1;
                    y+=sFlag;
                    d+=tAB;
                }
            }
        }
        else{//斜率>1
            if(kFlag==1){
                tA=2*b;
                d=2*b+a;
            }
            for(int i=0;i<abs(endPoint.y-startPoint.y);i++){
                if(d>=0)  {
                    pDC->SetPixel(x+1,y+sFlag,crColor);    //取点(x,y+1)
                    y+=sFlag;
                    x+=1;
                    d+=tAB;
                }
                else{//d<0
                    pDC->SetPixel(x,y+sFlag,crColor);      //取点(x+1,y+1)
                    y+=sFlag;
                    d+=tA;
                }
            }
        }
    }
    //此处省略特殊位置直线的算法代码
}
```

和数值微分法函数的保存方法一样，将中点画线函数也保存在 BasicGraph.h 文件中，以便在应用程序中直接引用。

中点画线算法在整个运算中都是整数运算，没有出现小数，因此占有的内存相对较少，也便于硬件实现。

3.1.4 Bresenham 画线算法

Bresenham 画线算法是计算机图形学领域使用最广泛的直线生成算法。在计算直线最佳逼近点的过程中，全部是整数运算，因此可以大幅提升计算速度。

算法原理如下：设直线从起点 $P_0(x_0,y_0)$ 到终点 $P_1(x_1,y_1)$，直线方程可表示为

$$y=kx+b$$

其中

$$k=\frac{y_1-y_0}{x_1-x_0}=\frac{\mathrm{d}y}{\mathrm{d}x},\quad b=y_0-kx_0$$

当直线斜率 $0<k\leqslant 1$ 时，直线在 x 方向步进，每次增加一个像素点，设当前直线的最佳逼近点是 (x_i,y_i)，则下一个最佳像素点的 x 坐标 $x_{i+1}=x_i+1$，y 坐标 $y_{i+1}=y_i$ 或者 $y_{i+1}=y_i+1$，y_{i+1} 到底取哪个值由当前直线上点的 y 坐标到 y_i 和 y_i+1 的距离大小而定，如图 3.1-8 所示。

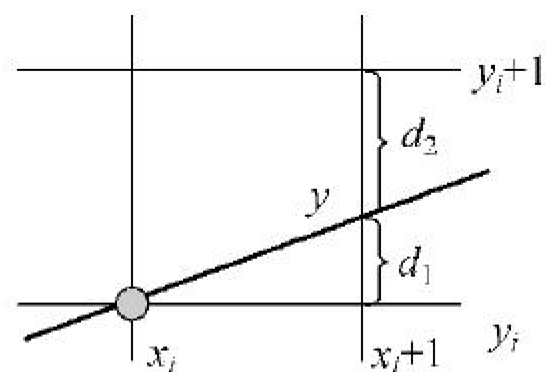


图 3.1-8　Bresenham 画线算法

$$y=k(x_i+1)+b$$
$$d_1=y-y_i$$
$$d_2=y_i+1-y$$

令

$$d_1-d_2=2y-2y_i-1=2(k(x_i+1)+b)-2y_i-1$$

当 $d_1-d_2>0$ 时，则

$$y_{i+1}=y_i+1$$

当 $d_1-d_2\leqslant 0$ 时，则

$$y_{i+1}=y_i$$

当 $\mathrm{d}x>0$ 时，用 $\mathrm{d}x$ 乘 d_1-d_2 不会改变 d_1-d_2 值的正负，因此不影响上述 y 的增量判断。并令 $p_i=(d_1-d_2)\mathrm{d}x$，同时，将 $k=\frac{\mathrm{d}y}{\mathrm{d}x}$代入，计算得

$$p_i=2x_i\mathrm{d}y-2y_i\mathrm{d}x+2\mathrm{d}y+(2b-1)\mathrm{d}x$$

当 $i=0$ 时，有

$$\begin{aligned}p_0&=2x_0\mathrm{d}y-2y_0\mathrm{d}x+2\mathrm{d}y+(2b-1)\mathrm{d}x\\&=2x_0\mathrm{d}y-2y_0\mathrm{d}x+2b\mathrm{d}x+2\mathrm{d}y-\mathrm{d}x\\&=2x_0\mathrm{d}y-2y_0\mathrm{d}x+2\left(y_0-\frac{\mathrm{d}y}{\mathrm{d}x}x_0\right)\mathrm{d}x+2\mathrm{d}y-\mathrm{d}x\\&=2x_0\mathrm{d}y-2y_0\mathrm{d}x+2(y_0\mathrm{d}x-\mathrm{d}yx_0)+2\mathrm{d}y-\mathrm{d}x\\&=2\mathrm{d}y-\mathrm{d}x\end{aligned}$$

当 $p_i>0$ 时，取点 (x_i+1,y_i+1)，则

$$p_{i+1}=2(x_i+1)\mathrm{d}y-2(y_i+1)\mathrm{d}x+2\mathrm{d}y+(2b-1)\mathrm{d}x=p_i+2\mathrm{d}y-2\mathrm{d}x$$

即增量为 $2\mathrm{d}y-2\mathrm{d}x$。

当 $p_i<0$ 时，取点 (x_i+1,y_i)，则

$$p_{i+1}=2(x_i+1)\mathrm{d}y-2y_i\mathrm{d}x+2\mathrm{d}y+(2b-1)\mathrm{d}x=p_i+2\mathrm{d}y$$

即增量为 $2\mathrm{d}y$。

上述是 $0<k\leqslant 1$ 的 Bresenham 算法，在编程实现时，和中点画线算法一样，也需要其他斜率时的情况。下面是一般位置直线的 Bresenham 算法参考代码（特殊位置直线算法代码省略）：

```
/**********************************************************************
BRESENHAM_Line: 直线的 Bresenham 算法函数
pDC: 显示器指针; startPoint: 起点; endPoint: 终点; crColor: 线的颜色
**********************************************************************/
```

```
void BRESENHAM_Line(CDC * &pDC,CPoint startPoint,CPoint endPoint,COLORREF crColor){
    if(endPoint.x!= startPoint.x&&endPoint.y!= startPoint.y){   //非特殊位置直线,一般直线
        int kFlag = 0;                                          //0:斜率≤1,1:斜率>1
        int sFlag = 1;                                          //斜率正负的标识
        //首先判断两个端点,通过交换使起点 x 小于终点 x
        if(startPoint.x > endPoint.x){
            CPoint pt = startPoint;
            startPoint = endPoint;
            endPoint = pt;
        }
        if(abs(endPoint.y - startPoint.y)> abs(endPoint.x - startPoint.x))
            kFlag = 1;                                          //如果斜率大于 1
        if(startPoint.y > endPoint.y)                           //如果斜率小于 0,则进行标识
            sFlag =- 1;
        int x,y;
        int dx,dy,p;
        if(sFlag ==- 1)
            endPoint.y = startPoint.y + (startPoint.y - endPoint.y);
        dx = endPoint.x - startPoint.x;
        dy = endPoint.y - startPoint.y;
        x = startPoint.x;
        y = startPoint.y;
        pDC -> SetPixel(x,y,crColor);
        if(kFlag == 0){                                         //斜率≤1
            p = 2 * dy - dx;
            for(int i = 0;i <(endPoint.x - startPoint.x);i++){
                if(p <= 0){
                    pDC -> SetPixel(x + 1,y,crColor);           //取点(x + 1,y)
                    x += 1;
                    p += 2 * dy;
                }
                else  {//p > 0
                    pDC -> SetPixel(x + 1,y + sFlag,crColor);   //取点(x + 1,y + 1)
                    x += 1;
                    y += sFlag;
                    p += 2 * dy - 2 * dx;
                }
            }
        }
        else{                                                   //斜率> 1
            p = 2 * dx - dy;
            for(int i = 0;i < abs(endPoint.y - startPoint.y);i++){
                if(p <= 0)  {
                    pDC -> SetPixel(x,y + sFlag,crColor);       //取点(x,y + 1)
                    y += sFlag;
                    p += 2 * dx;
                }
                else  {//p > 0
                    pDC -> SetPixel(x + 1,y + sFlag,crColor);   //取点(x + 1,y + 1)
                    y += sFlag;
                    x += 1;
```

```
                    p += 2 * dx - 2 * dy;
                }
            }
        }
    }
    //此处省略特殊位置直线的算法代码
}
```

和数值微分法函数以及中点画线函数的保存方法一样，将 Bresenham 画线函数也保存在 BasicGraph.h 文件中，以便在应用程序中直接引用。

3.1.5　图形程序设计及 VC++ 的橡皮筋和双缓存交互技术

直线、圆、椭圆等都是基本的图形元素。在进行图形应用时，会使用大量的图形元素，为了区分这些图形元素，非常有必要建立各种图形元素的数据结构或者图形类，这样，便于使用标准的格式。由于图形之间具有一些共同的属性，例如，颜色、线宽、线型等，所以，首先建立一个所有图形元素的基类。例如建立如下基类：

```
class CDraw{
    Public:
    CDraw(){}                          //构造函数
    COLORREF m_colorPen;               //颜色
    COLORREF m_colorBrush;             //填充颜色
    int m_lineWide;                    //线宽
    int m_lineType;                    //线型：实线，虚线，点划线，双点划线
};
```

则直线类可继承自该图形基类，再加入直线本身需要的两个端点变量即可。

为便于文件管理，将图形类放入一个单独的文件中，例如保存为文件 BasicClass.h，如下所示：

```
#ifndef BasicClass_
#define BasicClass_
class CDraw{
//....前述已经列出
};
//直线
class CLine:CDraw{
public:
    CLine& operator = (const CLine& ln){        //重载 = 等号操作符，实现直线赋值
        this->pt1 = ln.pt1;
        this->pt2 = ln.pt2;
        return *this;
    };
    bool operator == (const CLine& ln){         //重载 == 双等号操作符，判断两直线是否相同
        if(this->pt1 == ln.pt1&&this->pt2 == ln.pt2)
            return true;
        else
            return false;
```

```
    }
    CPoint pt1;
    CPoint pt2;
};
#endif
```

然后将该文件加入工程中,并在视图窗口类中使用时添加文件引用:

```
#include "BasicClass.h"
```

在视图窗口类中,建立直线集合:

```
CArray<CLine,CLine> m_line_array;
```

当拾取到两个点后,将拾取到的两个点生成直线,并加入直线集合中。例如在OnLButtonDown()函数中代码如下:

```
void CCGTest002View::OnLButtonDown(UINT nFlags, CPoint point) {
    if(m_iFlag == 0) {  //m_iFlag 为拾取直线的标识符,需在视图窗口类中定义
        if(this->ilstep == 0)  {      //ilstep 为拾取直线点的步骤,需在视图窗口类中定义
            lTmpPoint1 = point;       //lTmpPoint1 为直线第一点,在视图窗口类中定义
            this->ilstep = 1;                 //开始拾取直线第二点
        }
        else if(this->ilstep == 1){           //拾取直线第二点,并加入直线集合
        CLine line;
            line.pt1 = lTmpPoint1;
            line.pt2 = point;
            this->m_line_array.Add(line);     //加入直线集合
            this->ilstep = 0;                 //再开始拾取第一点
            Invalidate();                     //调用 OnDraw()函数
        }
    }
}
```

然后在OnDraw()函数中,将直线集合中的每条直线显示出来。为了使应用程序具有扩充性,以及便于代码管理和调用,可以将画线函数封装为函数DrawLines(),这样,在OnDraw()函数中,只需调用DrawLines()函数即可。调用DrawLines()函数的代码如下:

```
void CCGTest002View::DrawLines(CDC * pDC){
    if(this->m_line_array.GetSize()> 0){
        //画线
        for(int i = 0;i < this->m_line_array.GetSize();i++){
            line = this->m_line_array.GetAt(i);
            endPoint = line.pt2;
            startPoint = line.pt1;
            DrawLine(pDC,startPoint,endPoint); //调用直线生成算法函数
        }
    }
}
```

有多种直线生成算法,为了比较各种算法的效果,创建一个非模式对话框来选择不同的算法,并在非模式对话框中建立选择直线生成算法的三个单选按钮,如图3.1-9所示。

为这组单选按钮的第一个按钮建立 int 整型变量 m_iLine，其中，m_iLine＝0 代表选择的是数值微分法画线函数，m_iLine＝1 代表选择的是中点画线函数，m_iLine＝2 代表选择的是 Bresenham 画线函数。当选中不同的单选按钮时，需要对按钮单击消息 BN_CLICKED 增加消息处理函数，例如，数值微分法画线按钮的消息处理函数代码如下：

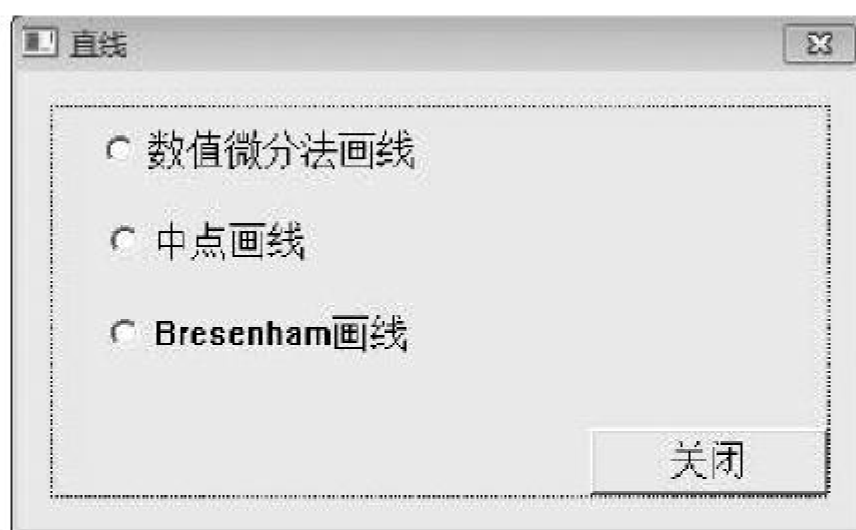


图 3.1-9 直线生成算法选择

```
void CLineDlg::OnRadio1() {
    UpdateData(TRUE);
    this->m_pView->Invalidate();
}
```

在视图窗口类中声明和使用该非模式对话框(非模式对话框的创建方法见 2.2 节)：

```
CLineDlg * m_lDlg;
```

在工具栏中设置一个画直线的图标，当画直线时，单击画直线工具栏，在其消息函数中，生成画线算法选择的非模式对话框 m_lDlg：

```
void CCGTest002View::OnLine() {
    this->m_iFlag = 0;                              //画直线
    this->ilstep = 0;                               //设置开始拾取直线第一个点
    if(this->m_lineDlg->GetSafeHwnd() == NULL){     //生成画线算法选择对话框
        this->m_lineDlg->Create();
    }
    else
        this->m_lineDlg->ShowWindow(TRUE);
}
```

在直线生成算法函数 DrawLine()中，根据非模式对话框中算法的选项，选择不同的直线生成算法，代码如下：

```
void CCGTest002View::DrawLine(CDC * pDC,CPoint startPoint,CPoint endPoint){
    if(this->m_lineDlg->m_iLine == 0)  //m_lineDlg 为画线设置的非模式对话框
        DDA_Line(pDC,startPoint,endPoint,RGB(255,0,0));
    else if(this->m_lineDlg->m_iLine == 1)
        MIDPOINT_Line(pDC,startPoint,endPoint,RGB(255,0,0));
    else
        BRESENHAM_Line(pDC,startPoint,endPoint,RGB(255,0,0));
}
```

执行程序，直线生成效果如图 3.1-10 所示。

上述通过 OnDraw()函数生成的是位置确定的最终图形，如果在位置最终确定之前希望看到图形的动态变化效果，例如，随着鼠标在视图窗口的移动，能够实时动态地生成图形，并进行观察，就需要用到 VC6.0 的交互式绘图技术。VC6.0 的绘图功能中有一个“异或”的绘图特性，即在屏幕上用异或的模式画图形，相同的位置重新画一次此图形，则会在屏幕

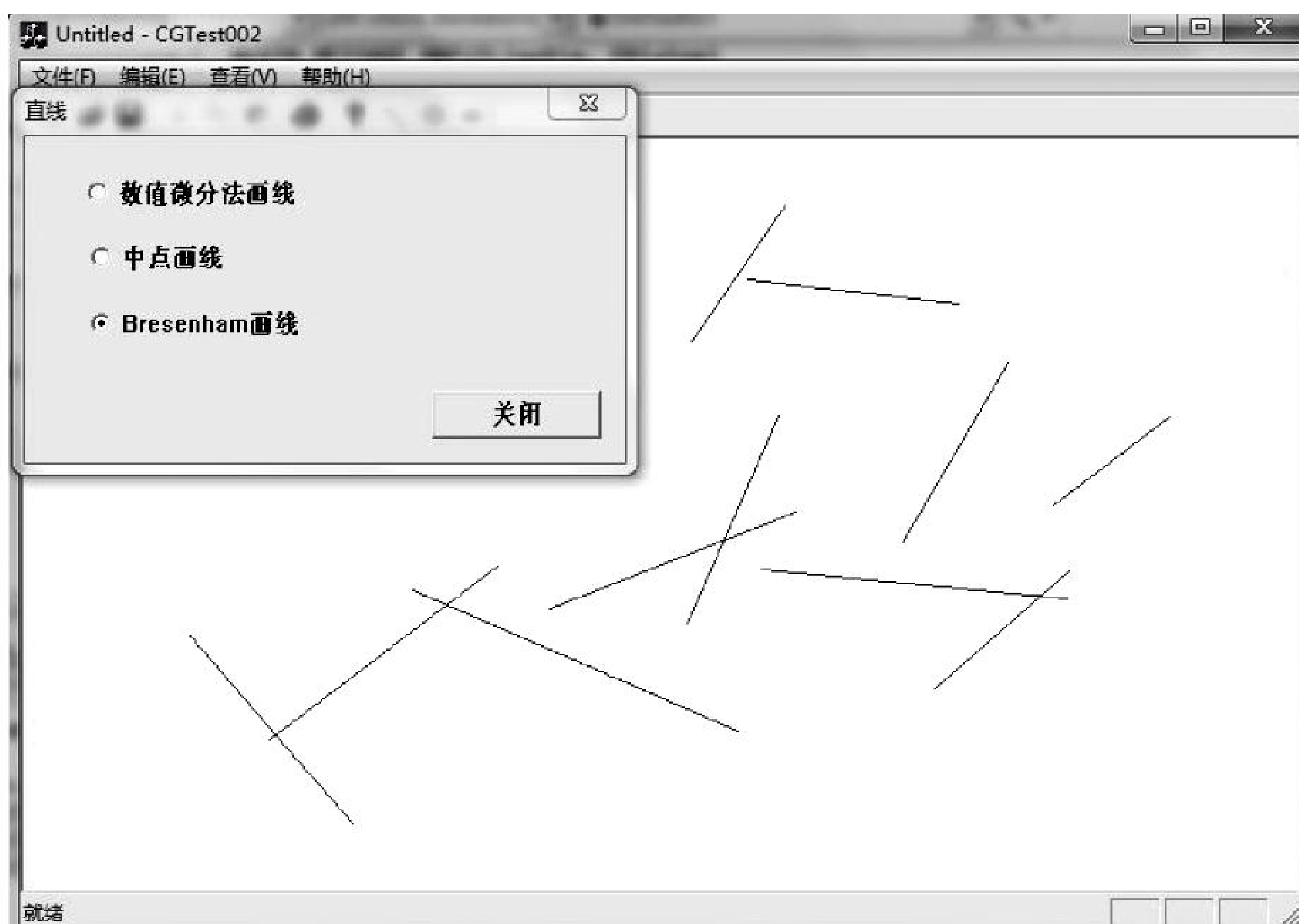


图 3.1-10　直线扫描转换算法比较

上擦除上一次所绘制的内容。利用这种特性画直线时，直线就像橡皮筋一样，随着鼠标在屏幕中位置的移动而移动，长短也随之变化，因此，又称之为橡皮筋技术。

橡皮筋技术一般在鼠标移动时使用，因此，在鼠标移动的映射函数 OnMouseMove()中实现。其实现的代码非常简单，在绘制新位置图形之前，只需调用 pDC->SetROP2(R2_NOT)，然后把上次相同位置的图形再绘制一次，即可擦除上次绘制图形，然后，再绘制新图形。具体代码如下所示：

```
void CCGTest002View::OnMouseMove(UINT nFlags, CPoint point) {
    if(m_iFlag == 0){                                    //绘制直线
        if(this -> ilstep == 1)  {
            CDC * pDC = GetDC();                         //获得图形显示设备指针
            pDC -> SetROP2(R2_NOT);
            DrawLine(pDC,lTmpPoint1,lTmpPoint2);         //绘制上次的图形,进行擦除
            DrawLine(pDC,lTmpPoint1,point);              //绘制新位置的图形
            lTmpPoint2 = point;
            ReleaseDC(pDC);                              //释放图形显示设备指针
        }
    }
}
```

将拾取点的函数修改如下：

```
void CCGTest002View::OnLButtonDown(UINT nFlags, CPoint point) {
    if(m_iFlag == 0){  //m_iFlag 为拾取直线的标识符,需在视图窗口类中定义
        if(this -> ilstep == 0){  //ilstep 为拾取直线点的步骤,需在视图窗口类中定义
```

```
            lTmpPoint1 = point;              //lTmpPoint1 为第一点,需在视图窗口类中定义
            lTmpPoint2 = point;              //lTmpPoint2 为第二点,需在视图窗口类中定义
            this->ilstep = 1;                //开始拾取第二个点
        }
        else if(this->ilstep == 1){          //拾取第二个点,则生成直线
            CLine line;
            line.pt1 = lTmpPoint1;
            line.pt2 = point;
            this->m_line_array.Add(line);    //加入直线集合中
            CDC * pDC = GetDC();
            DrawLine(pDC,lTmpPoint1,point);  //对现有两点画线
            ReleaseDC(pDC);
            this->ilstep = 0;                //再开始拾取第二个点
            //Invalidate();                  //删除调用 OnDraw()函数
        }
    }
}
```

通过上面的代码修改处理,即可在视图窗口移动鼠标时实时地动态生成直线,并在两个端点完全确定时再将直线加入直线集合中。橡皮筋技术不仅适用于直线的生成,其他图形如圆、椭圆的生成也可以使用该技术,从而获得动态的图形生成效果。

OnDraw()函数是绘制图形的主要函数,在程序中使用 Invalidate()函数时,会调用 OnDraw()函数,将图形在显示设备上重新再绘制一遍,当应用程序的窗口发生变化时,例如,窗口移动、窗口缩放或者窗口滚动,也会调用 OnDraw()函数,重绘图形。OnDraw()函数首先用背景色将显示区域清除,然后,再重绘,由于背景色往往与绘图内容反差很大,这样短时间内背景色与显示图形交替出现,使得窗口看起来有闪烁的感觉。为了避免这种闪烁的现象,可以采用双缓存技术来解决。

双缓存技术的原理可以这样形象地理解：把电脑屏幕看作一块黑板,首先在内存环境中建立一个虚拟的黑板,然后在这块黑板上绘制复杂的图形,等图形全部绘制完毕的时候,再一次性把内存中绘制好的图形复制到屏幕上。也就是除了在屏幕上进行图形显示以外,在内存中也绘制图形,把要显示的图形先在内存中绘制好,然后再一次性将内存中的图形一个点一个点地覆盖到屏幕上去(这个过程非常快,因为是非常规整的内存复制)。这样在内存中绘图时,随便用什么反差大的背景色进行清除都不会闪,因为看不到。当粘贴到屏幕上时,因为内存中最终的图形与屏幕显示图形差别很小(如果没有运动,当然就没有差别),这样看起来就不会闪。

因为背景色是在内存中绘制的,所以,没必要再在屏幕上清除背景色,这样就避免了闪烁。通过重载 WM_ERASEBKGND 消息函数,去掉屏幕背景绘制。代码如下：

```
BOOL CCGTest002View::OnEraseBkgnd(CDC * pDC) {
    return TRUE;                           //直接返回 TRUE
    //return CView::OnEraseBkgnd(pDC);     //去掉屏幕背景清除
}
```

OnDraw()函数的代码修改如下：

```
void CCGTest002View::OnDraw(CDC * pDC){
```

```
    CCGTest002Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    //获取显示窗口的大小
    RECT rect;
    GetClientRect(&rect);
    int w = rect.right;
    int h = rect.bottom;
    //获取显示区域原点在绘图区的坐标
    CPoint OrgPt;                          // = pDC - > GetScrollPosition()//滚动窗口时使用
    OrgPt.x = 0;                           //OrgPt.x;
    OrgPt.y = 0;                           //OrgPt.y;
    int Xoffset = OrgPt.x;
    int Yoffset = OrgPt.y;
    CDC memDC;
    CBitmap memBitmap;
    //创建与屏幕显示兼容的内存显示设备和位图
    memDC.CreateCompatibleDC(NULL);
    memBitmap.CreateCompatibleBitmap(pDC,w,h);          //位图的大小同屏幕显示区
    memDC.SelectObject(&memBitmap);                     //把位图选入设备环境
    memDC.FillSolidRect(0,0,w,h,RGB(255,255,255));      //设置背景色为白色
    DrawLines(&memDC);                                  //此部分为绘图部分
    //将内存显示设备直接复制到屏幕显示设备的显示区
    pDC - > BitBlt(Xoffset,Yoffset,w,h,&memDC,0,0,SRCCOPY);
    memBitmap.DeleteObject();                           //删除位图
    memDC.DeleteDC();                                   //删除内存
}
```

由于双缓存技术是首先在内存中画图形,然后再一次性地将图片复制到显示屏幕上,因此,不适用于图形实时调试的开发环境。在调试图形程序时,当需要逐步调试图形生成过程,并观察生成效果时,建议转换为直接在显示设备上生成图形。

3.2 圆的扫描转换

3.2.1 圆的扫描转换概述

圆的扫描转换就是在屏幕像素点阵中确定一组最佳逼近于圆的像素点,并用指定的颜色显示出来。

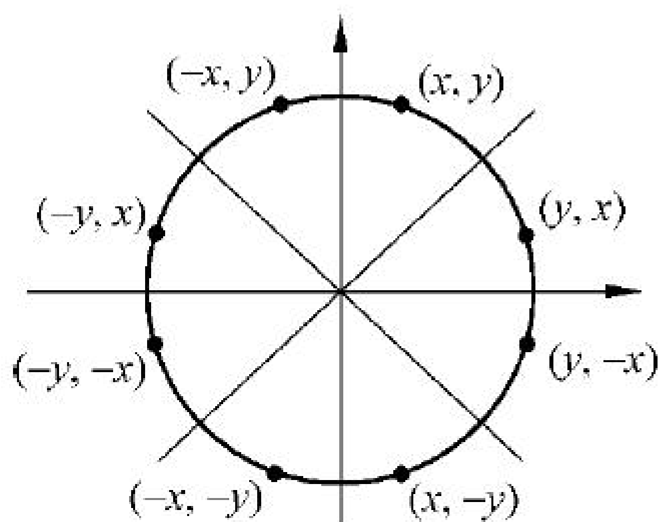


图 3.2-1　圆对称生成点

当圆心在坐标原点时,圆的方程为 $x^2+y^2=R^2$;当圆心不在坐标原点时,首先获得圆心通过坐标原点的圆的最佳像素逼近点,再对像素逼近点进行坐标平移即可。

由于圆具有对称性,在进行扫描转换时,只需迭代生成八分之一圆的最佳像素逼近点,圆的其他部分通过简单的坐标对称就可以直接得到,如图 3.2-1 所示。假设已知圆上的一个点 (x,y),则利用对称性,可得到圆上其他的七个

点$(x,-y)$，$(-x,y)$，$(-x,-y)$，(y,x)，$(y,-x)$，$(-y,x)$，$(-y,-x)$。

3.2.2　中点画圆算法

中点画圆法和直线的中点画线法的原理类似，首先，构造圆的隐式方程

$$F(x,y)=x^2+y^2-R^2$$

圆上的点$F(x,y)=0$，圆外的点$F(x,y)>0$，圆内的点$F(x,y)<0$，在直角坐标第一象限，从点$(0,R)$到$\left(\frac{R}{\sqrt{2}},\frac{R}{\sqrt{2}}\right)$的八分之一圆弧内，$x$值按像素单位递增，$y$值逐渐减少。构造判别式

$$d=F(M)=F(x_i+1,y_i-0.5)=(x_i+1)^2+(y_i-0.5)^2-R^2$$

若$d\geqslant 0$，则该点在圆外，如图3.2-2(a)所示，最佳像素逼近点应取(x_i+1,y_i-1)，则构造下一个像素点的判别式为

$$\begin{aligned}d_1&=F(M)=F(x_i+2,y_i-1.5)=(x_i+2)^2+(y_i-1.5)^2-R^2\\&=d+2(x_i-y_i)+5\end{aligned}$$

即增量为$2(x_i-y_i)+5$。

若$d<0$，则该点在圆内，如图3.2-2(b)所示，最佳像素逼近点应取(x_i+1,y_i)，则构造下一个像素点的判别式为

$$d_1=F(M)=F(x_i+2,y_i-1.5)=(x_i+2)^2+(y_i-0.5)^2-R^2=d+2x_i+3$$

即增量为$2x_i+3$。

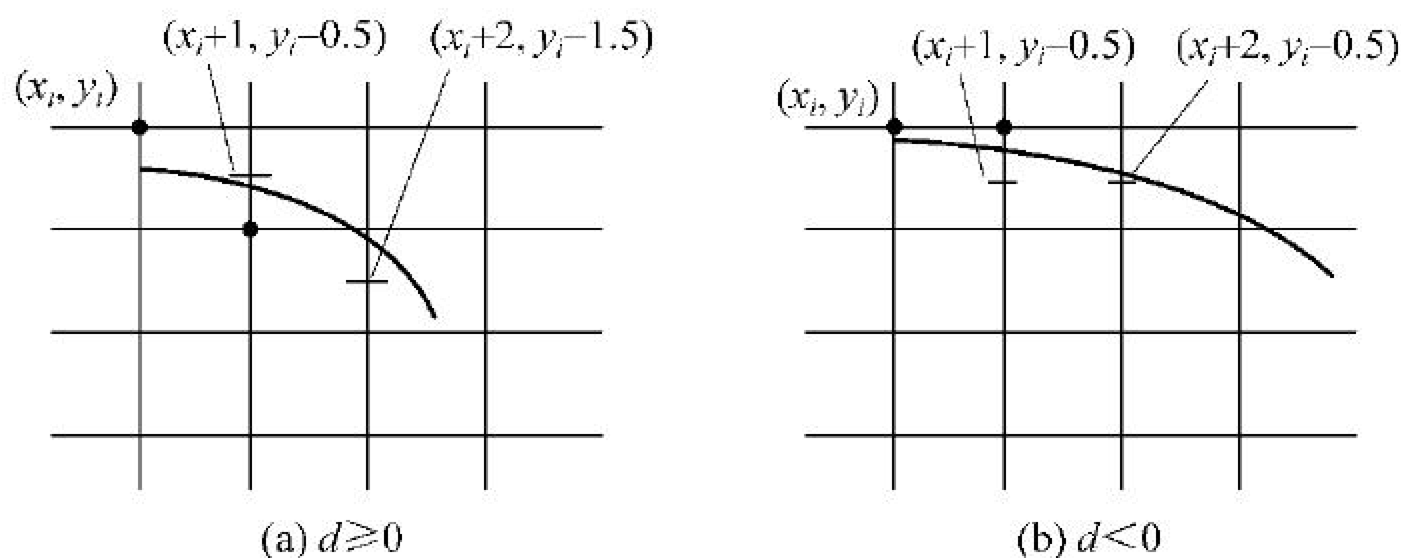


图3.2-2　圆的判别式

初始的构造点为$(1,R-0.5)$，判别式为

$$d_0=F(M)=F(1,R-0.5)=1^2+(R-0.5)^2-R^2=1.25-R$$

在上述方法中，初始判别式的计算中出现了小数，可以通过乘以4将小数转换为整数，则迭代算法可改进为

$$d_0=5-4R$$

$d\geqslant 0$，取(x_i+1,y_i-1)，$d_1=d+8(x_i-y_i)+20$

$d<0$，取(x_i+1,y_i)，$d_1=d+8x_i+12$

中点画圆算法的函数参考代码如下：

```
/*********************************************************************
Mid_Circle: 中点画圆算法函数
```

```
pDC: 显示器指针; cPt: 圆心; R: 半径; crColor: 颜色
**********************************************************************/
void Mid_Circle(CDC * pDC,CPoint cPt,int R,COLORREF crColor){
    int x,y,d;
    x = 0;
    y = R;
    d = 5 - 4 * R;
    pDC->SetPixel((cPt.x + x),(cPt.y + y),crColor);
    pDC->SetPixel((cPt.x + x),(cPt.y - y),crColor);
    pDC->SetPixel((cPt.x + y),(cPt.y + x),crColor);
    pDC->SetPixel((cPt.x - y),(cPt.y + x),crColor);
    while(x<=y)  {
        if(d>=0)  {
            x += 1;
            y -= 1;
            d += 8 * (x - y) + 20;
        }
        else  {
            x += 1;
            d += 8 * x + 12;
        }
        pDC->SetPixel((cPt.x + x),(cPt.y + y),crColor);
        pDC->SetPixel((cPt.x - x),(cPt.y + y),crColor);
        pDC->SetPixel((cPt.x + x),(cPt.y - y),crColor);
        pDC->SetPixel((cPt.x - x),(cPt.y - y),crColor);
        pDC->SetPixel((cPt.x + y),(cPt.y + x),crColor);
        pDC->SetPixel((cPt.x - y),(cPt.y + x),crColor);
        pDC->SetPixel((cPt.x + y),(cPt.y - x),crColor);
        pDC->SetPixel((cPt.x - y),(cPt.y - x),crColor);
    }
}
```

3.2.3 Bresenham 画圆算法

圆的 Bresenham 算法和直线的 Bresenham 算法类似，在迭代时，也是通过判断圆上的点与附近像素点距离的差值正负，来获得最佳像素逼近点。在迭代时，也是只需计算直角坐标第一象限从 $(0,R)$ 到 $\left(\frac{R}{\sqrt{2}},\frac{R}{\sqrt{2}}\right)$ 的八分之一圆弧最佳逼近点，其他点通过对称得到。

设圆的当前最佳逼近点为 (x_i,y_i)，则当 $x_{i+1}=x_i+1$ 时，$y^2=R^2-(x_i+1)^2$，下一个最佳逼近点为 (x_i+1,y_i) 或者 (x_i+1,y_i-1)，如图 3.2-3 所示。则构造下面的判别式：

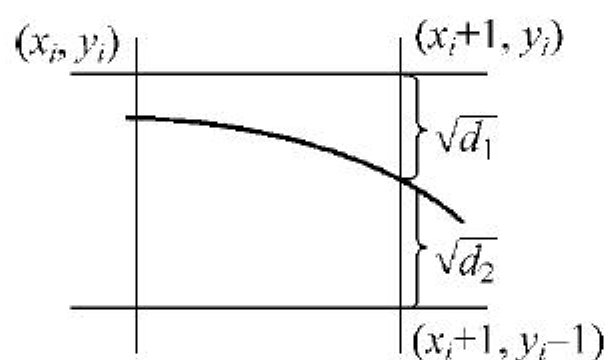


图 3.2-3 Bresenham 画圆

$$d_1 = y_i^2 - y^2 = y_i^2 - (R^2 - (x_i + 1)^2)$$

$$d_2 = y^2 - (y_i - 1)^2 = R^2 - (x_i + 1)^2 - (y_i - 1)^2$$

令 $p_i = d_1 - d_2$，则

$$p_i = 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2R^2$$

若 $p_i > 0$，则取点 (x_i+1,y_i-1)，再下一个像素点的判别式为

$$p_{i+1} = 2(x_i+1+1)^2+(y_i-1)^2+(y_i-1-1)^2-2R^2 = p_i+4(x_i-y_i)+10$$

即增量是 $4(x_i-y_i)+10$。

若 $p_i \leqslant 0$,则取点 (x_i+1, y_i),再下一个像素点的判别式为

$$p_{i+1} = 2(x_i+1+1)^2+y_i^2+(y_i-1)^2-2R^2 = p_i+4x_i+6$$

即增量是 $4x_i+6$。

在 $(0,R)$ 点时可得判别式的初值:

$$p_0 = 3-2R$$

Bresenham 画圆算法的函数代码参考如下:

```
/**********************************************************************
Bresenham_Circle: 中点画圆算法函数
pDC: 显示器指针; cPt: 圆心; R: 半径; crColor: 颜色
**********************************************************************/
void Bresenham_Circle(CDC * pDC,CPoint cPt,int R,COLORREF crColor){
    int x,y,p;
    x = 0;
    y = R;
    p = 3 - 2 * R;
    pDC->SetPixel((cPt.x + x),(cPt.y + y),crColor);
    pDC->SetPixel((cPt.x + x),(cPt.y - y),crColor);
    pDC->SetPixel((cPt.x + y),(cPt.y + x),crColor);
    pDC->SetPixel((cPt.x - y),(cPt.y + x),crColor);
    while(x <= y){
        if(p >= 0)  {
            x += 1;
            y -= 1;
            p += 4 * (x - y) + 10;
        }
        else  {
            x += 1;
            p += 4 * x + 6;
        }
        pDC->SetPixel((cPt.x + x),(cPt.y + y),crColor);
        pDC->SetPixel((cPt.x - x),(cPt.y + y),crColor);
        pDC->SetPixel((cPt.x + x),(cPt.y - y),crColor);
        pDC->SetPixel((cPt.x - x),(cPt.y - y),crColor);
        pDC->SetPixel((cPt.x + y),(cPt.y + x),crColor);
        pDC->SetPixel((cPt.x - y),(cPt.y + x),crColor);
        pDC->SetPixel((cPt.x + y),(cPt.y - x),crColor);
        pDC->SetPixel((cPt.x - y),(cPt.y - x),crColor);
    }
}
```

通过比较 Bresenham 画圆算法代码和中点画圆算法,可以看到,虽然两种算法的基本原理和推导过程不同,但是算法的构造迭代公式却基本上是相同的,因此两种算法的计算速度也是相同的。

在应用程序中生成圆时，可以按照如下方法建立圆的结构类：

```
class CCircle:CDraw{
public:
    CPoint Opt;                                   //圆心
    int rLength;                                  //半径
};
```

在视图窗口类中建立圆的集合：CArray<CCircle,CCircle> m_circle_array;，利用鼠标在视图窗口拾取圆心，并设计或者输入圆的半径值，也可以类似直线生成程序，利用橡皮筋技术实时生成，并比较中点画圆算法和 Bresenham 画圆算法，如图 3.2-4 所示。圆生成程序的流程和代码可参考 3.1 节"直线的扫描转换"中的相关内容实现。

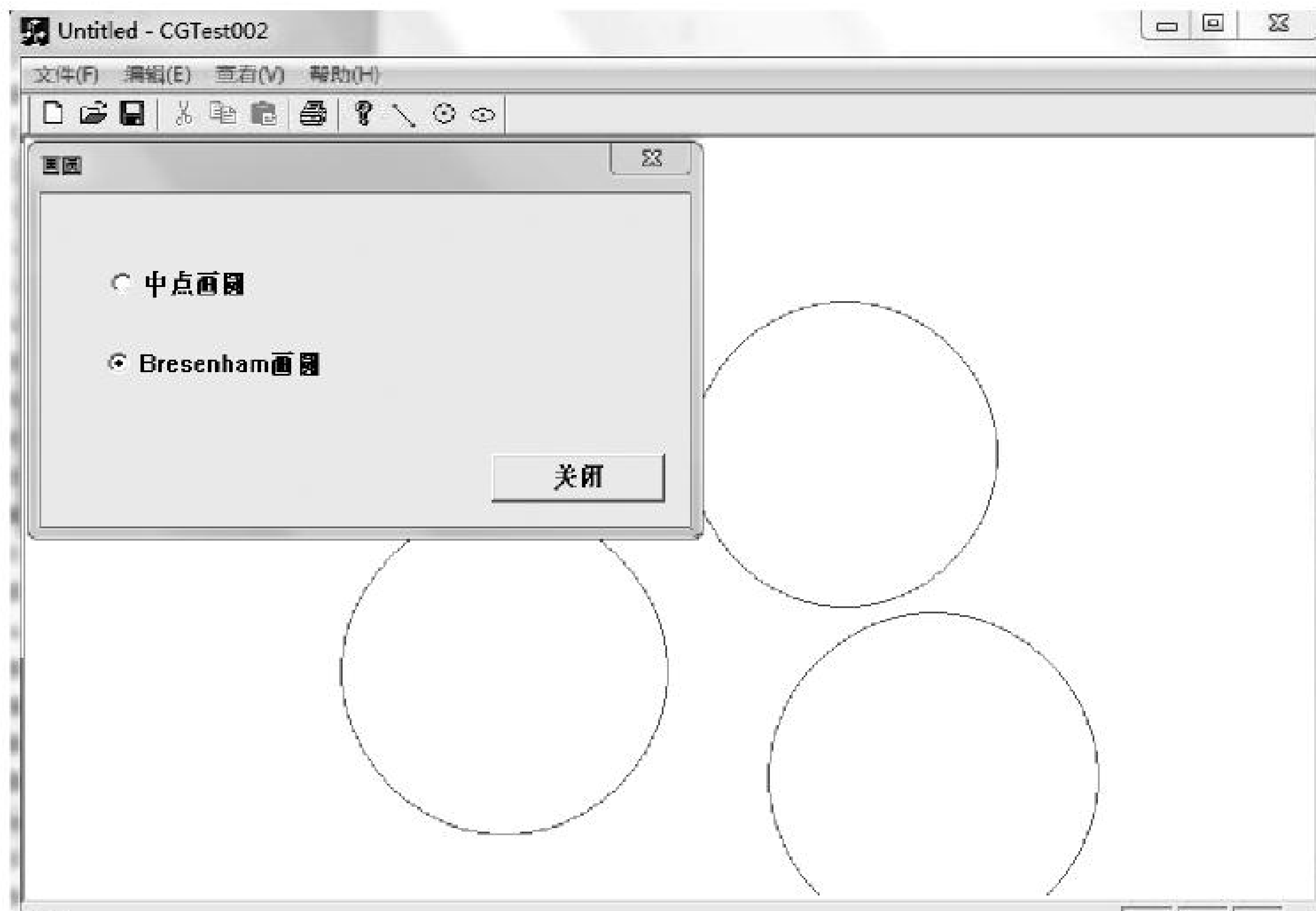


图 3.2-4　圆的生成

3.2.4　圆弧段的扫描转换

圆弧段的扫描转换是指仅将圆中的任意一段圆弧利用一组屏幕像素点最佳逼近，并用指定的颜色显示出来。圆弧段的扫描转换也可以利用圆的扫描转换方法的中点画圆算法或者 Bresenham 算法的思路实现，但是，由于圆弧段在圆上的位置具有任意性，圆弧段的长度和两个端点的位置也是任意的，所以，圆弧段的扫描转换实现方法比整个圆在标准的八分之一圆弧上通过镜像生成的扫描转换方法复杂。

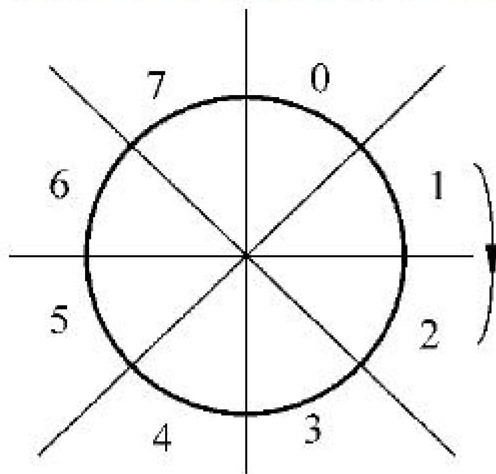


图 3.2-5　圆分区

圆上两点之间对应的圆弧段有两个，为了具有确定性，设其中一个点为圆弧的起点，另外一个点为圆弧的终点，从起点到终点顺时针方向的这段圆弧为所求的圆弧段。

首先，利用轴线、45°及 135°线作镜像线将圆划分成如下八个区域，按顺时针方向分别用数字 0～7 进行编号，如图 3.2-5 所示。整圆的中点画圆算法和 Bresenham 画圆算法都是首先在圆的 0 号区域进行迭代生成，其他区域的点利用镜像对称性生成。点之间

的对称性可以利用坐标变换矩阵表示，设0号区域的矩阵为 $\boldsymbol{T}(0)=\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$，则其他区域的坐标变换矩阵分别为 $\boldsymbol{T}(1)=\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$，$\boldsymbol{T}(2)=\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}$，$\boldsymbol{T}(3)=\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$，由于对称性，矩阵 $\boldsymbol{T}(4)$、$\boldsymbol{T}(5)$、$\boldsymbol{T}(6)$、$\boldsymbol{T}(7)$ 可以分别通过将矩阵 $\boldsymbol{T}(0)$、$\boldsymbol{T}(1)$、$\boldsymbol{T}(2)$、$\boldsymbol{T}(3)$ 乘以 -1 获得到。

设圆上0区域的一个点 (x_0, y_0) 用向量表示为 $[x_0 \quad y_0]$，则其在其他区域的镜像点——例如 i 区域的镜像点 (x, y)——的向量为该点向量乘以 i 区域的坐标变换矩阵得到：

$$[x \quad y] = [x_0 \quad y_0] \cdot \boldsymbol{T}(i)$$

相反，当圆上点在其他区域，例如 k 区域的一个点 (x_1, y_1)，如要获得其在0区域的镜像对应点，可通过该点的向量乘以 k 区域坐标变换矩阵的逆矩阵计算得出：

$$[x \quad y] = [x_1 \quad y_1] \cdot \boldsymbol{T}^{-1}(k)$$

同理，圆上其他某个区域内的圆弧段在0区域上也对应了一段圆弧，那么，只要在0区域上对与其对应的圆弧段的最佳像素逼近点利用中点画线算法或者其他算法进行迭代生成，但是对像素点不进行颜色显示，而是通过坐标变换矩阵仅在要求的圆弧段上的像素显示颜色，这样，就实现了其他圆弧区域内的任意圆弧段的扫描转换。这种方法的关键是要准确找到圆弧段在0区域所对应的位置以及起始和终止点。

设圆弧段的起点为 (x_s, y_s)，终点为 (x_e, y_e)，圆弧所在圆的圆心为 (x_c, y_c)，半径为 R。圆弧段的起点 (x_s, y_s) 和终点 (x_e, y_e) 的位置关系有两种情况，第一种情况是起点和终点在同一个区域内，第二种情况是起点和终点不在同一个区域，即横跨几个区域。这两种情况需要分别考虑。

当起点和终点在同一个区域内时，扫描转换方法即为上述的圆上其他某个区域内的圆弧的生成方法：用起点和终点的向量乘以所在区域的坐标变换矩阵的逆矩阵即得两点在0区域的对应点，对0区域的两点之间的圆弧进行迭代生成，然后转换到起点和终点所在的区域显示。设起点 (x_s, y_s) 的对应点是 (x_0, y_0)，终点 (x_e, y_e) 的对应点是 (x_1, y_1)，如图3.2-6所示。如果镜像转换后圆弧的方向发生了变化，那么应将起点和终点互换。

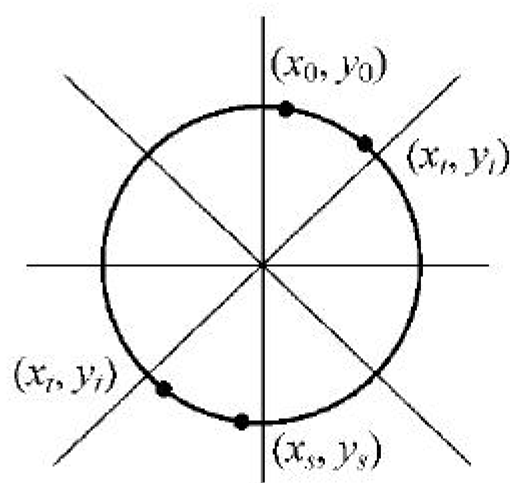


图3.2-6 起点终点在一个区域

由于圆弧段不一定是完整的八分之一圆弧，利用算法迭代生成时，计算初始判别式值的构造点不再是 $(0, R)$，而是起点 (x_0, y_0)。例如中点画圆算法的初始判别式的值为

$$\begin{aligned} d_0 &= F(M) = F(x_0+1, y_0-0.5) = (x_0+1)^2 + (y_0-0.5)^2 - R^2 \\ &= 1.25 + 2x_0 - y_0 \end{aligned}$$

和圆的算法处理方法相同，将判别式乘以4，把小数变成整数：

$$d_0 = F(M) = (1.25 + 2x_0 - y_0) \times 4 = 5 + 8x_0 - 4y_0$$

同样，迭代的终止条件也要修改为圆弧画到终点 (x_1, y_1) 时结束。利用上述算法即可绘制出整个部分都在一个区域内的圆弧段。

当起点和终点横跨几个区域时，即不在同一个区域的情况，为了生成圆弧，首先将圆弧段分成三个部分分别处理，这三部分分别是起点所在区域的圆弧段、终点所在区域的圆弧段

以及起点和终点之间隔着的几个八分之一区域的圆弧段，如图 3.2-7 所示。这样划分后，起点和终点所在的圆弧段就符合同在一个区域的情况，这时，圆弧段的另外一个端点在镜像线上，利用上述一个区域内的圆弧段的扫描转换方法就可以生成。在具体实现时需要注意，同样的圆弧段处在不同的区域时，在 0 区域对应的圆弧段的位置是不同的，同一个区域的起点和终点的圆弧段在 0 区域对应的圆弧段的位置也是不同的。设起点(x_s, y_s)所在的区域是 I_s，在 0 区域的对应点是(x_0, y_0)。

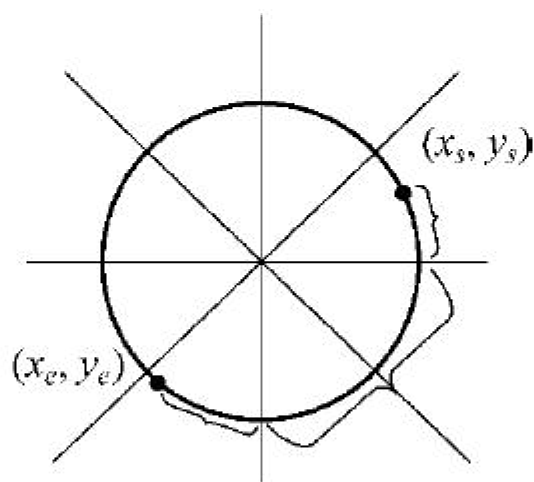


图 3.2-7　起点终点不在一个区域

当 $I_s \in [0,2,4,6]$时，起点部分的圆弧段在 0 区域对应的是$\left[(x_s, y_s), \left(\frac{R}{\sqrt{2}}, \frac{R}{\sqrt{2}}\right)\right]$段圆弧；

当 $I_s \in [1,3,5,7]$时，起点部分的圆弧段在 0 区域对应的是$[(0,R),(x_s, y_s)]$段圆弧。

同样区域的终点部分的圆弧在 0 区域对应的圆弧段和起点部分的圆弧在 0 区域对应的圆弧段恰好相反。

圆弧起点和终点之间隔着的中间几个区域部分的圆弧是 0 区域的八分之一圆弧段的整数倍，利用中点画圆算法或者 Bresenham 画圆算法将 0 区域的圆弧段$\left[(0,R), \left(\frac{R}{\sqrt{2}}, \frac{R}{\sqrt{2}}\right)\right]$迭代生成后，通过其他几个区域的坐标变换矩阵获得对应区域的像素点，即可得到这几个部分的圆弧段。

起点部分的圆弧、终点部分的圆弧以及中间部分的圆弧合在一起就是所需生成的圆弧段。

由于起点和终点在圆上的位置是任意的，起点和终点所在的区域也是任意的，但是，圆弧段是指顺时针方向从起点到终点的圆弧，为了使区域具有连贯性，应该保证终点所在区域的编号不小于起点区域的编号。设终点所在的区域是 I_e，对 I_e 作如下处理：

当 $I_e < I_s$ 时，$I_e = I_e + 8$。

在计算所在区域的坐标变换矩阵或者逆矩阵时，将 I_e 与 8 求余数：$I_e = I_e \% 8$，即得准确的区域编号。

上述圆弧段实现方法的函数代码参考如下：

```
/**********************************************************************
DrawCircleArc: 圆弧段生成函数
pDC: 显示器指针; cPt: 圆心; R: 半径; sPt: 起点; ePt: 终点; crColor: 颜色
**********************************************************************/
void DrawCircleArc(CDC * pDC,CPoint cPt,int arcR,CPoint sPt,CPoint ePt,COLORREF crColor){
    CPoint s_Pt,e_Pt;
    //圆心移动到原点
    s_Pt.x = sPt.x - cPt.x;
    s_Pt.y = sPt.y - cPt.y;
    e_Pt.x = ePt.x - cPt.x;
    e_Pt.y = ePt.y - cPt.y;
    //第一步,首先查找两个端点所在的区域编号
    int T[2][2],T1[2][2];                //镜像转换矩阵
    CPoint Pt0,Pt1;                      //圆点处镜像后的起始终止点
```

```
    int iIs = FindIndexOfArc(s_Pt);
    int iIe = FindIndexOfArc(e_Pt);
    if(iIe < iIs) iIe += 8;
    if(iIs == iIe){                       //起始点和终止点在一个区域
        //首先镜像到 0 号弧段,生成线再镜像到圆弧段内画线
        GetMatrixT(T,iIs);                //获得在 0 弧段的镜像矩阵
        GetTtoPt(T,s_Pt,Pt0,1);           //生成 0 弧段的对应点
        GetTtoPt(T,e_Pt,Pt1,1);
        if(Pt0.x < Pt1.x)                 //从 pt0 画到 pt1
            SetArc(pDC,cPt,arcR,3,Pt0,Pt1,T,crColor);
        else                              //从 pt1 画到 pt0
            SetArc(pDC,cPt,arcR,3,Pt1,Pt0,T,crColor);
    }
    else {                                //起始终止点不在一个区段,则分三部分分别生成
        //1.起始点段圆弧
        GetMatrixT(T,iIs);                //获得在 0 弧段的镜像矩阵
        GetTtoPt(T,s_Pt,Pt0,1);           //生成 0 弧段的对应点
        if(iIs % 2 == 0)                  //从起点到八分之一镜像线生成圆弧
            SetArc(pDC,cPt,arcR,2,Pt0,Pt1,T,crColor);
        else                              //从(0,R)到起点之间的圆弧
            SetArc(pDC,cPt,arcR,1,Pt0,Pt1,T,crColor);
        //2.中间圆弧段
        for(int i = iIs + 1;i < iIe;i++){
            GetMatrixT(T1,i % 8);         //获得 i 弧段的镜像矩阵
            SetArc(pDC,cPt,arcR,0,Pt0,Pt1,T1,crColor);  //画八分之一圆弧
        }
        //3.终止点圆弧段
        GetMatrixT(T,iIe);                //获得在 0 弧段的镜像矩阵
        GetTtoPt(T,e_Pt,Pt1,1);           //生成 0 弧段的对应点
        if(iIe % 2 == 1)                  //从终止点到八分之一镜像线生成圆弧
            SetArc(pDC,cPt,arcR,2,Pt1,Pt0,T,crColor);
        else                              //从(0,R)到终止点之间的圆弧
            SetArc(pDC,cPt,arcR,1,Pt1,Pt0,T,crColor);
    }
return;
}
```

上述代码中,查找圆弧端点所在的区域编号的函数如下:

```
int FindIndexOfArc(CPoint Pt){            /* cPt:圆心,Pt:判断的圆弧点 */
    int x = Pt.x;
    int y = Pt.y;
    if(x > 0&&y > 0&&x < y) return 0;
    if(x > 0&&y > 0&&x > y) return 1;
    if(x > 0&&y < 0&&x > - y) return 2;
    if(x > 0&&y < 0&&x < - y) return 3;
    if(x < 0&&y < 0&& - x < - y) return 4;
    if(x < 0&&y < 0&& - x > - y) return 5;
    if(x < 0&&y > 0&& - x > y) return 6;
    if(x < 0&&y > 0&& - x < y) return 7;
    return - 1;
}
```

其中，计算区域坐标变换矩阵的函数为：

```
void GetMatrixT(int T[2][2],int iI){          /*T: 坐标变换矩阵,iI: 区域编号*/
    if(iI%8==0){
        T[0][0]=1;T[0][1]=0;
        T[1][0]=0;T[1][1]=1;
    }
    else if(iI%8==1){
        T[0][0]=0;T[0][1]=1;
        T[1][0]=1;T[1][1]=0;
    }
    else if(iI%8==2){
        T[0][0]=0;T[0][1]=-1;
        T[1][0]=1;T[1][1]=0;
    }
    else if(iI%8==3){
        T[0][0]=1;T[0][1]=0;
        T[1][0]=0;T[1][1]=-1;
    }
    else if(iI%8==4){
        T[0][0]=-1;T[0][1]=0;
        T[1][0]=0;T[1][1]=-1;
    }
    else if(iI%8==5){
        T[0][0]=0;T[0][1]=-1;
        T[1][0]=-1;T[1][1]=0;
    }
    else if(iI%8==6){
        T[0][0]=0;T[0][1]=1;
        T[1][0]=-1;T[1][1]=0;
    }
    else if(iI%8==7){
        T[0][0]=-1;T[0][1]=0;
        T[1][0]=0;T[1][1]=1;
    }
}
```

利用坐标变换矩阵生成区域的像素点和变换逆矩阵获得在0区域的对应点的函数为：

```
void GetTtoPt(int T[2][2],CPoint& Pt,CPoint& retnPt,int InverseFlag=0){  /*T: 坐标变换矩
阵; retnPt: 返回点; InverseFlag: =1-矩阵求逆*/
    if(InverseFlag==1){                       //逆阵
    //主对角线元素互换并除以行列式的值,副对角线元素变号并除以行列式的值
        int T1[2][2],tmp,tmp1;
        tmp=T[0][0];
        T1[0][0]=T[1][1];
        T1[1][1]=tmp;
        T1[1][0]=T[1][0]*(-1);
        T1[0][1]=T[0][1]*(-1);
        tmp1=T[0][0]*T[1][1]-T[1][0]*T[0][1];
        retnPt.x=(Pt.x*T1[0][0]+Pt.y*T1[1][0])/tmp1;
```

```
        retnPt.y = (Pt.x * T1[0][1] + Pt.y * T1[1][1])/tmp1;
    }
    else  {
        retnPt.x = (Pt.x * T[0][0] + Pt.y * T[1][0]);
        retnPt.y = (Pt.x * T[0][1] + Pt.y * T[1][1]);
    }
    return;
}
```

下面是代码中用到的具体一个圆弧段的生成函数，其中迭代生成点的方法利用的是中点画圆算法的原理：

```
/* pDC: 显示器指针; cPt: 圆心; ArcType =0:八分之一圆弧, =1: 从(0,R)到 Pt_S 的圆弧, =2:从
Pt_s 到 x=y(八分之一镜像线)的圆弧, =3:Pt_s 到 Pt_e 的圆弧,Pt_s: 起点,Pt_e: 终点; T: 坐标变
换矩阵; crColor: 颜色 */
void SetArc(CDC *pDC,CPoint cPt,int R,int ArcType,CPoint &Pt_s,CPoint &Pt_e,int T[][2],
COLORREF crColor){
    int x,y,d;
    CPoint p0,p1;
    if(ArcType==0||ArcType==1){
        x=0;
        y=R;
        d=5-4*R;
    }
    else  {
        x=Pt_s.x;
        y=Pt_s.y;
        d=8*x-4*y+5;
    }
    //镜像到原始点所在圆弧区间上生成
    p0.x=x;
    p0.y=y;
    GetTtoPt(T,p0,p1);
    pDC->SetPixel((cPt.x+p1.x),(cPt.y+p1.y),crColor);
    while(1)  {                              //迭代生成条件
        if(ArcType==0||ArcType==2){          //生成到八分之一镜像线处
            if(x>y) break;
        }
        else if(ArcType==1){                 //生成从(0,R)到 Pt_S 的圆弧
            if(x>Pt_s.x) break;
        }
        else if(ArcType==3){                 //生成到 Pt_e 的圆弧
            if(x>Pt_e.x) break;
        }
        if(d>=0)  {
            x+=1;
            y-=1;
            d+=8*(x-y)+20;
        }
        else  {
            x+=1;
```

```
                d += 8 * x + 12;
            }
            p0.x = x;
            p0.y = y;
            GetTtoPt(T,p0,p1);
            pDC -> SetPixel((cPt.x + p1.x),(cPt.y + p1.y),crColor);
        }
}
```

上述算法生成圆弧段的效果图如图 3.2-8 所示。

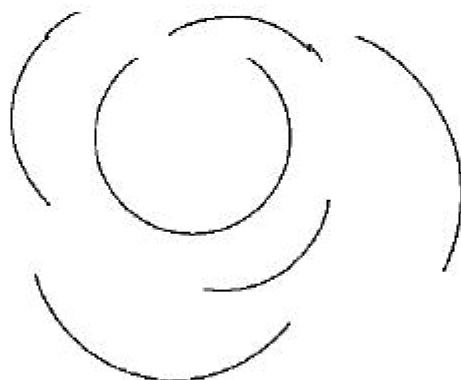


图 3.2-8 圆弧段扫描转换

除了利用圆心、起点和终点确定圆弧外，利用其他方式也可以确定一个圆弧段。例如，圆上三个点确定一个圆弧，已知圆心、起点和圆弧角也可以确定一个圆弧。利用其他方式得到的圆弧可以通过计算转换成圆心、起点、终点的圆弧段表示。由于圆弧段是整圆的一部分，圆弧段也可以通过圆的裁剪生成，为了使圆和圆弧具有统一的表示方法，对圆的结构类作如下进一步的改进：

```
class CCircle:CDraw{
public:
    CCircle(){
    Type = 0;
    DirectFlag = 0;   }
    CPoint Opt;                         //圆心
    int rLength;                        //半径
    int Type;                           //圆的类型 0:整圆,1: 圆弧
    int DirectFlag;                     //圆的方向 0: 顺时针,1: 逆时针
    CPoint Spt;                         //第一个点,也指起点
    CPoint Ept;                         //第二个点,也指终点
    CPoint Tpt;                         //三点弧时,第三个点
};
```

3.3 椭圆的扫描转换

中点画圆法可以推广到一般二次曲线的扫描转换，例如椭圆的扫描转换。图 3.3-1 所示通过坐标原点的椭圆方程为

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

其中 a 为沿 x 轴方向的长半轴长度，b 为沿 y 轴方向的短半轴长度。由于椭圆具有对称性，因此只需讨论第一象限部分的椭圆的生成，其他三部分通过对称实现。在处理这段椭圆弧时，需要从弧上斜率为−1(即法向矢量为 **1**)的点进一步把它分为上下两部分，如图 3.3-2 所示，法向量 $\mathbf{N}>\mathbf{1}$，即点在上一部分时，在 x 方向进行单位增量；法向量 $\mathbf{N}<\mathbf{1}$，即点在下一部分时，在 y 方向进行单位增量。

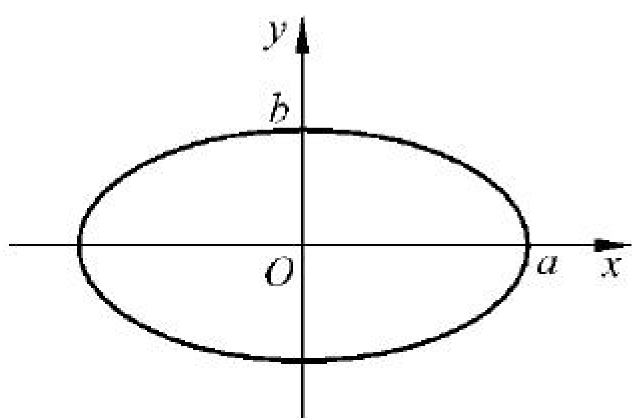


图 3.3-1 椭圆

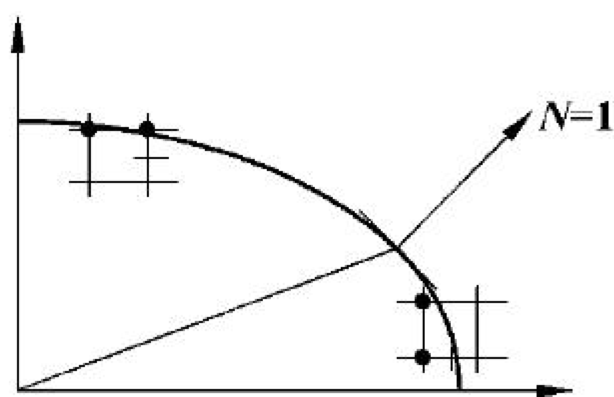


图 3.3-2 椭圆法向量 $N=1$

与中点画圆算法类似，首先在做选择的两个像素的中点处构造判别式，并根据判别式的正负来确定最佳逼近像素，然后，再构造下一个判别式。

假设当前最佳像素逼近点为 (x_i,y_i)，在 x 方向进行单位增量，构造中点为 $(x_i+1,y_i-0.5)$，则判别式为

$$d=F(x_i+1,y_i-0.5)=b^2(x_i+1)^2+a^2(y_i-0.5)^2-a^2b^2$$

若 $d<0$，构造的中点在椭圆内，则下一个最佳像素点为 (x_i+1,y_i)。再构造下一个中点为 $(x_i+2,y_i-0.5)$，则判别式为

$$\begin{aligned}d_1&=F(x_i+2,y_i-0.5)=b^2(x_i+2)^2+a^2(y_i-0.5)^2-a^2b^2\\&=d+b^2(2x_i+3)\end{aligned}$$

即增量为 $b^2(2x_i+3)$。

若 $d\geqslant 0$，构造的中点在椭圆外，则下一个最佳像素点为 (x_i+1,y_i-1)。再构造下一个中点为 $(x_i+2,y_i-1.5)$，则判别式为

$$\begin{aligned}d_1&=F(x_i+2,y_i-1.5)=b^2(x_i+2)^2+a^2(y_i-1.5)^2-a^2b^2\\&=d+b^2(2x_i+3)+a^2(-2y_i+2)\end{aligned}$$

即增量为 $b^2(2x_i+3)+a^2(-2y_i+2)$。

从起点 $(0,b)$ 处开始构造初始中点 $(1,b-0.5)$，对应的初始判别式为

$$d_0=F(1,b-0.5)=b^2+a^2(b-0.5)^2-a^2b^2=b^2+a^2(-b+0.25)$$

上述迭代是法向量 $\mathbf{N}\geqslant 1$ 时的算法。当 $\mathbf{N}<1$ 时，在 y 方向单位增量。根据微分知识，椭圆上点 (x,y) 处的法向量 $\mathbf{N}$ 为

$$\mathbf{N}(x,y)=\frac{\partial F}{\partial x}\boldsymbol{i}+\frac{\partial F}{\partial y}\boldsymbol{j}=2b^2x\boldsymbol{i}+2a^2y\boldsymbol{j}$$

其中 $\boldsymbol{i}$ 和 $\boldsymbol{j}$ 分别为该点在 x 方向和 y 方向的单位向量。当 $\mathbf{N}\geqslant 1$ 时，法向量 y 分量较大；$\mathbf{N}<1$ 时，法向量 x 分量较大。因此 $\mathbf{N}\geqslant 1$ 时，即为 $2b^2x\leqslant 2a^2y\Rightarrow b^2x\leqslant a^2y$，由于构造的中点和椭圆上的点非常接近，可以认为构造的中点 $(x_i+1,y_i-0.5)$ 也满足

$$b^2(x_i+1)\leqslant a^2(y_i-0.5)$$

在构造下一个中点时，如果上面的不等式不再成立，则说明这时椭圆上点的法向量 $\mathbf{N}<1$，这时开始在 y 方向单位增量。

与在 x 方向的单位增量计算迭代类似，构造 $\mathbf{N}<1$ 时的中点和判别式。假设当前最佳像素逼近点为 (x_i,y_i)，在 y 方向单位增量，构造中点为 $(x_i+0.5,y_i-1)$，则判别式为

$$d=F(x_i+0.5,y_i-1)=b^2(x_i+0.5)^2+a^2(y_i-1)^2-a^2b^2$$

若 $d<0$，构造的中点在椭圆内，下一个最佳像素点为 (x_i+1,y_i-1)。再构造下一个中点为 $(x_i+1.5,y_i-2)$，则判别式为

$$d_1 = F(x_i + 1.5, y_i - 2) = b^2(x_i + 1.5)^2 + a^2(y_i - 2)^2 - a^2b^2$$
$$= d + b^2(2x_i + 2) + a^2(-2y_i + 3)$$

即增量为 $b^2(2x_i+2)+a^2(-2y_i+3)$。

若 $d \geqslant 0$，构造的中点在椭圆外，下一个最佳像素点为 (x_i, y_i-1)。再构造下一个中点为 $(x_i+0.5, y_i-2)$，则判别式为

$$d_1 = F(x_i + 0.5, y_i - 2) = b^2(x_i + 0.5)^2 + a^2(y_i - 2)^2 - a^2b^2$$
$$= d + a^2(-2y_i + 3)$$

即增量为 $a^2(-2y_i+3)$。

当 $y_i=0$ 时，迭代终止。

上述的椭圆中点算法函数代码参考如下：

```
/*********************************************************************
MidPt_Ellipse: 中点画椭圆算法函数
pDC: 显示器指针; cPt: 圆心; a: 第一个半轴; b: 第二个半轴; crColor: 颜色
*********************************************************************/
void MidPt_Ellipse(CDC * pDC,CPoint cPt,int a,int b,COLORREF crColor){
    int x,y;
    double d;
    x = 0;
    y = b;
    d = b * b + a * a * ( - b + 0.25);           //初始值
    pDC->SetPixel((cPt.x + x),(cPt.y + y),crColor);
    pDC->SetPixel((cPt.x + x),(cPt.y - y),crColor);
    pDC->SetPixel((cPt.x - x),(cPt.y + y),crColor);
    pDC->SetPixel((cPt.x - x),(cPt.y - y),crColor);
    while(b * b * (x + 1)< a * a * (y - 0.5)){    //法向量大于 1
        if(d >= 0)  {
            x += 1;
            y -= 1;
            d += b * b * (2 * x + 3) + a * a * ( - 2 * y + 2);
        }
        else  {
            x += 1;
            d += b * b * (2 * x + 3);
        }
        pDC->SetPixel((cPt.x + x),(cPt.y + y),crColor);
        pDC->SetPixel((cPt.x + x),(cPt.y - y),crColor);
        pDC->SetPixel((cPt.x - x),(cPt.y + y),crColor);
        pDC->SetPixel((cPt.x - x),(cPt.y - y),crColor);
    }
    while(y > 0){                                 //法向量小于 1
        if(d >= 0)  {
            y -= 1;
            d += a * a * ( - 2 * y + 3);
        }
        else  {
            x += 1;
            y -= 1;
```

```
            d+=a*a*(-2*y+3)+b*b*(2*x+2);
        }
        pDC->SetPixel((cPt.x+x),(cPt.y+y),crColor);
        pDC->SetPixel((cPt.x+x),(cPt.y-y),crColor);
        pDC->SetPixel((cPt.x-x),(cPt.y+y),crColor);
        pDC->SetPixel((cPt.x-x),(cPt.y-y),crColor);
    }
}
```

在应用程序中生成椭圆时,可建立如下的椭圆类:

```
class CEllipse:CDraw{
    public:
    CPoint Opt;                              //椭圆中心点
    int a,b;                                 //椭圆的两个半轴长
};
```

在视图窗口类建立椭圆的集合:`CArray<CEllipse,CEllipse> m_ellipse_array;`,通过窗口拾取椭圆的中心点和设计两个半轴的长度,或者直接通过对话框输入对应值,生成的椭圆如图3.3-3所示。椭圆生成程序的流程和代码可参考3.1节"直线的扫描转换"中的相关内容实现。

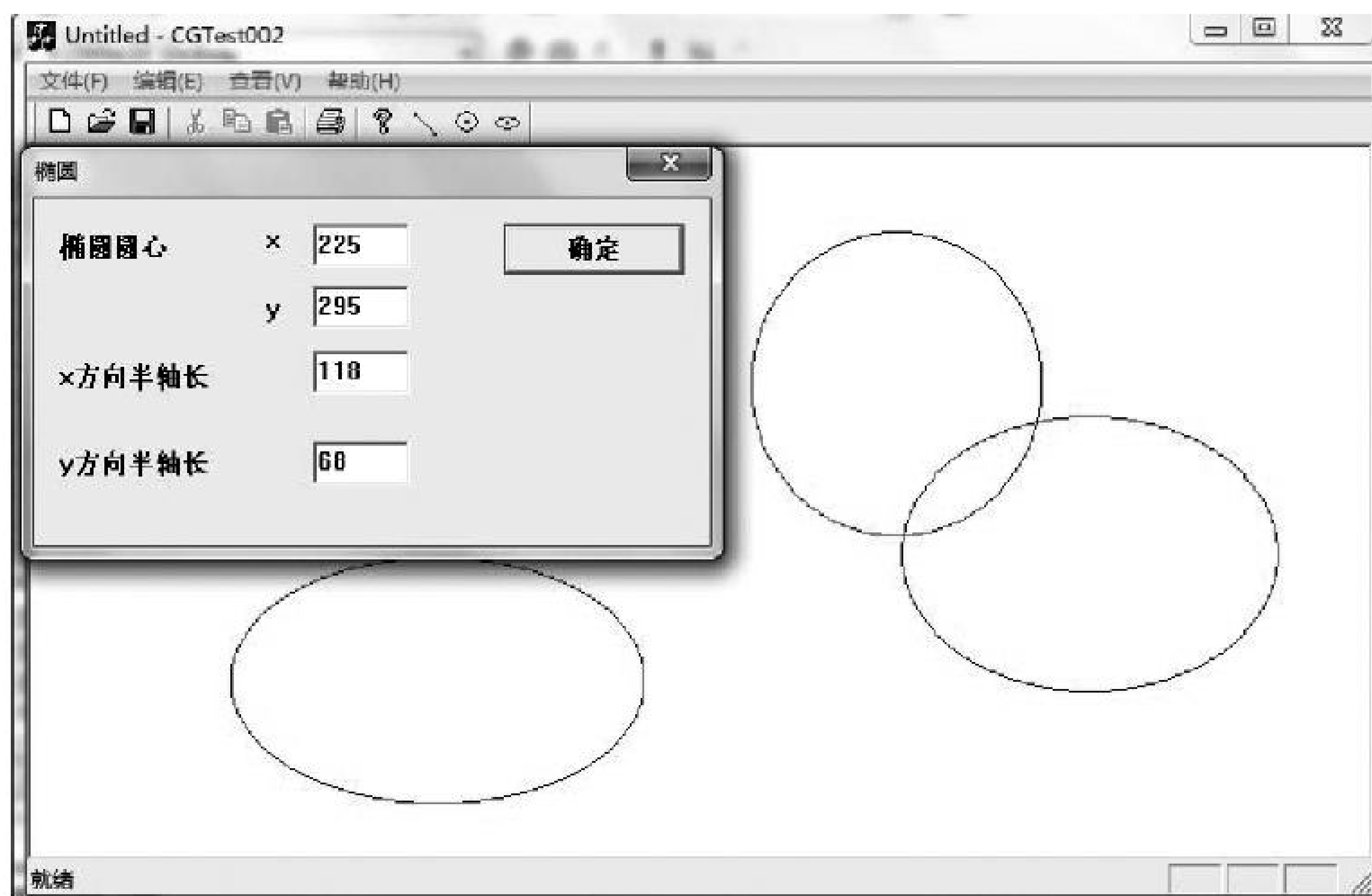


图3.3-3 椭圆的生成

3.4 多边形的扫描转换及区域填充

3.4.1 多边形的扫描转换

多边形是指由首尾相接的直线段组成的封闭图形。形成的这个封闭区域是多边形的内部,封闭区域外侧是多边形的外部,多边形的顶点即由直线段的端点组成,组成多边形边的

直线段之间不能发生交叉现象。根据形状特点，多边形可以分为下面三种类型：

(1) 凸多边形

凸多边形指任意两个顶点的连线都在多边形内部的多边形，如图 3.4-1(a)所示。

(2) 凹多边形

凹多边形指任意两个顶点的连线有不在多边形内部的多边形，如图 3.4-1(b)所示。

(3) 带内环的多边形

带内环的多边形指多边形内部还嵌套有多边形，如图 3.4-1(c)所示。这时的多边形内部是指最外侧的多边形和嵌套的多边形之间的区域，如果有多个嵌套的多边形，则要求嵌套的多边形之间不能交叉，也不能再嵌套。嵌套的多边形又称为内环，最外侧的多边形称为外环。

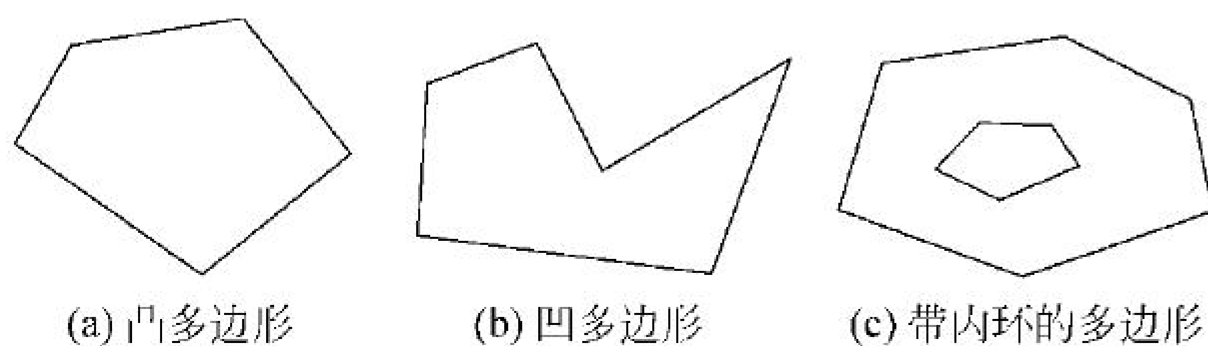


图 3.4-1　多边形的类型

计算机中多边形的表示方法有两种。

(1) 顶点表示法

用多边形的顶点序列来表示多边形，如图 3.4-2 所示。顶点表示法的特点是直观、几何意义强，易于图形变换，而且占用内存较少，但是不能区分多边形的内外部。

(2) 点阵表示法

用位于多边形内部的像素点集来表示多边形，如图 3.4-3 所示。点阵表示法的特点是可以明确多边形的内部区域，并将内部区域的像素点显示成要求的颜色，但是缺少多边形顶点的几何信息。

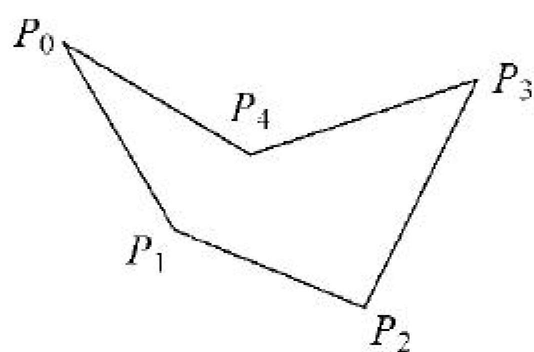


图 3.4-2　顶点表示法

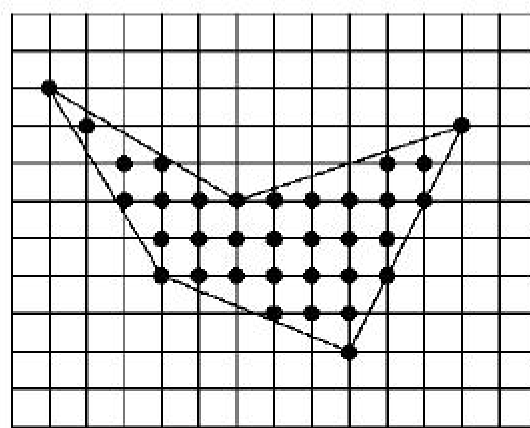


图 3.4-3　点阵表示法

多边形的扫描转换就是把多边形从顶点表示转换为点阵表示。多边形的扫描转换又称为多边形的区域填充，即从多边形的顶点信息出发，求出多边形的内部区域的像素点，并用要求的颜色显示出来。

扫描线算法是一种常用的多边形区域填充算法，其基本原理是按扫描顺序，计算每条扫描线与多边形的相交区间，再用要求的颜色显示这些区间的像素，即完成填充工作。区间的端点可以通过计算扫描线与多边形边界线的交点获得，如图 3.4-4 所示。

扫描线算法的核心：须按 x 的递增顺序排列扫描线与多边形边线的交点序列，并形成

区间。算法步骤如下。

(1) 确定多边形所占有的最大扫描线数：得到顶点的最小和最大 y 值(y_{min} 和 y_{max})。

(2) 从 $y=y_{min}$ 到 $y=y_{max}$ 每次用一条扫描线进行填充。

(3) 对一条扫描线填充的过程可分为四个步骤。

① 求交：计算扫描线与多边形各个边的交点。

② 排序：将所有交点按 x 值的递增顺序排序。

③ 交点配对：将交点按递增顺序两两配成区间对，例如第一点和第二点、第三点和第四点分别组成区间对。交点对之间构成的区间就是多边形内部的像素点。

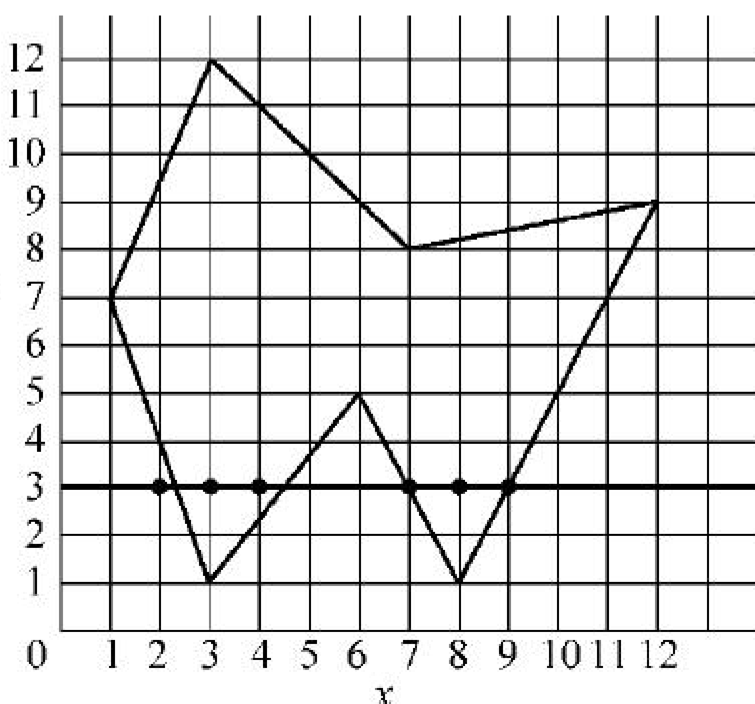


图 3.4-4　扫描线多边形填充算法

④ 区间填色：将区间对内的像素点设置成要求的颜色。

扫描线算法的原理简单，实施时需要提高算法的准确度和填充效率。算法准确度主要是指当扫描线通过多边形的顶点时，由于顶点是相邻两个边的共有点，那么同一个交点会计算两次，这时会存在交点数的取舍问题。如图 3.4-5 所示，当 y=1,5,7,8,9,12 时，会通过多边形的顶点，顶点即交点，只有取合适的交点数，才能正确形成区间对。

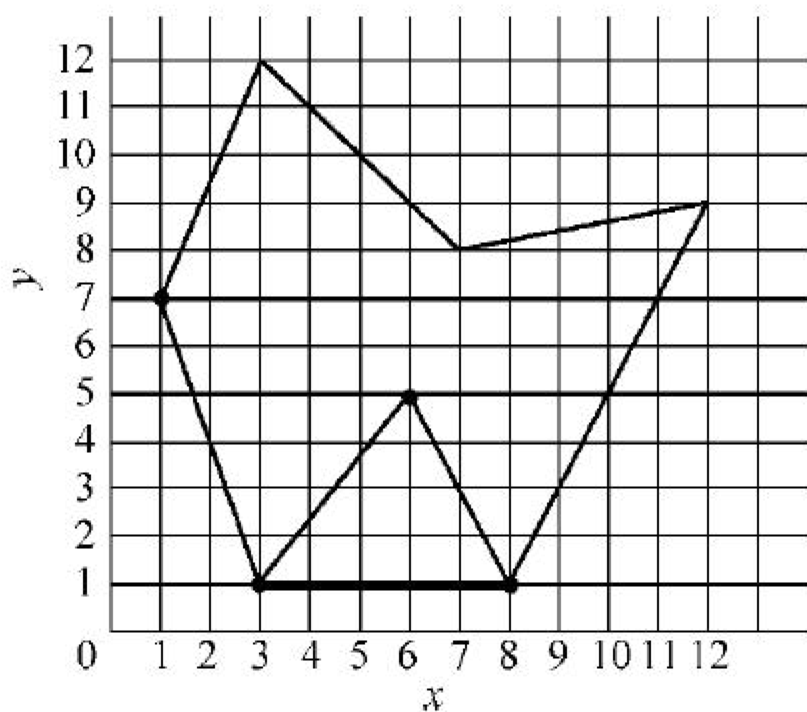


图 3.4-5　交点取舍

解决方法是，若共享顶点的两条边分别在扫描线的两侧，则交点只算一个；若共享顶点的两条边在扫描线的同一侧，这时交点记为零个或两个。可以通过检查共享顶点的两条边的另外两个端点的 y 值，按这两个 y 值中大于交点 y 值的个数来决定交点数取 0、1 或者 2。若共享顶点的两条边分别在扫描线的两侧，在代码中编程时，可以通过将每条边的 y_{max} 减少一个单位，或者将 y_{min} 增加一个单位，实现只计算一个交点，如图 3.4-6 所示。

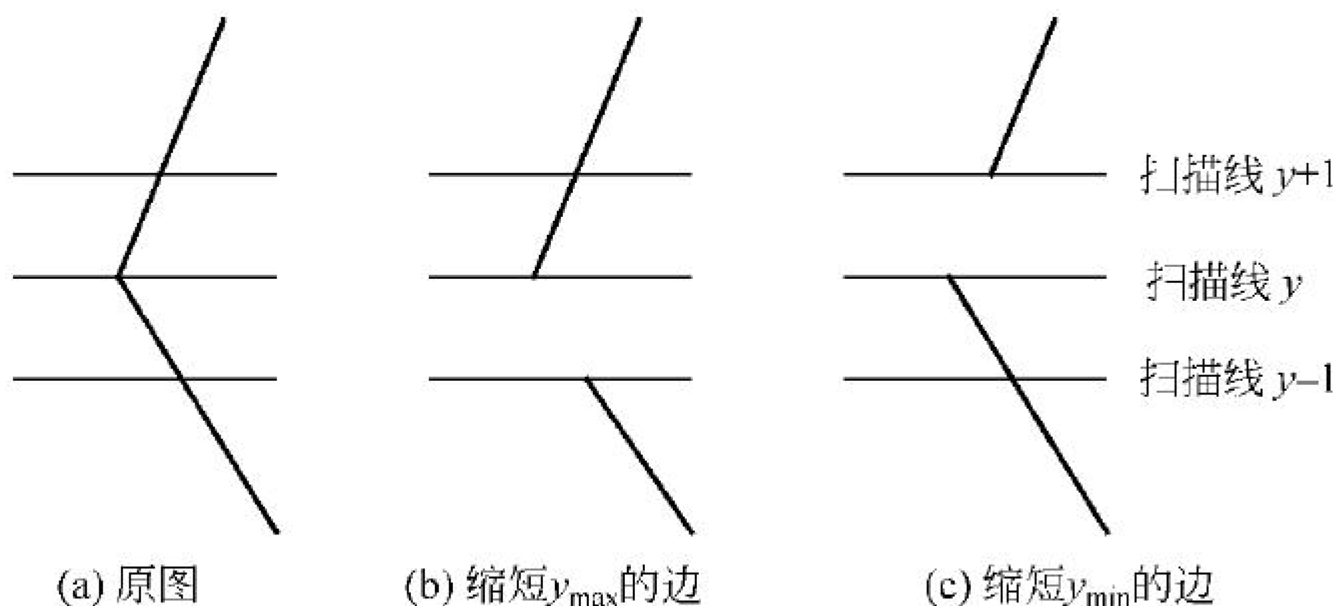


图 3.4-6　共享顶点的两个边的处理

在应用程序中实现多边形扫描转换时，首先需要构造多边形。根据多边形的类型，多边形的结构类如下：

```
class CPolyLine:CDraw{
public:
    CPolyLine(){                                //多边形函数
        in_num = 0;                             //内环数量最初为 0
    }
    ~CPolyLine()  {                             //多边形析构函数,即设置该多边形为空
        this->m_PolyLine_array_Out.RemoveAll();
        for(int i = 0;i < in_num;i++){
            this->m_PolyLine_array_in[i].RemoveAll();
        }
        in_num = 0;                             //内环数量最初为 0
    }
    CPolyLine& operator =(const CPolyLine& polyline){   //=号重载,多边形赋值
        this->m_PolyLine_array_Out.RemoveAll();
        this->m_PolyLine_array_Out.Append(polyline.m_PolyLine_array_Out);
        this->in_num = polyline.in_num;
        for(int i = 0;i < polyline.in_num;i++)  {
            this->m_PolyLine_array_in[i].RemoveAll();
            this->m_PolyLine_array_in[i].Append(polyline.m_PolyLine_array_in[i]);
        }
        return *this;
    }
    bool operator ==(const CPolyLine& polyline){   //==重载,判断两多边形是否相同
        if(this == &polyline) return true;
        for(int i = 0;i < this->m_PolyLine_array_Out.GetSize();i++){   //外环判断
if((this->m_PolyLine_array_Out.GetAt(i) == polyline.m_PolyLine_array_Out.GetAt(i)) ==
false)
                return false;
        }
        if(this->in_num!= polyline.in_num) //内环判断
            return false;
        for(i = 0;i < this->in_num;i++){
            for(int k = 0;k < this->m_PolyLine_array_in[i].GetSize();k++){
if((this->m_PolyLine_array_in[i].GetAt(k) == polyline.m_PolyLine_array_in[i].GetAt(k)) ==
false)
                    return false;
            }
        }
        return true;
    }
    CArray<CLine,CLine> m_PolyLine_array_Out;           //多边形外环
    CArray<CLine,CLine> m_PolyLine_array_in[10000];     //多边形内环
    int in_num;                                         //内环数量
};
```

通过鼠标拾取屏幕点,并连接前后点和首尾相接,获得多边形的外环和内环,利用多边形数组得到多组多边形,如图 3.4-7 所示。

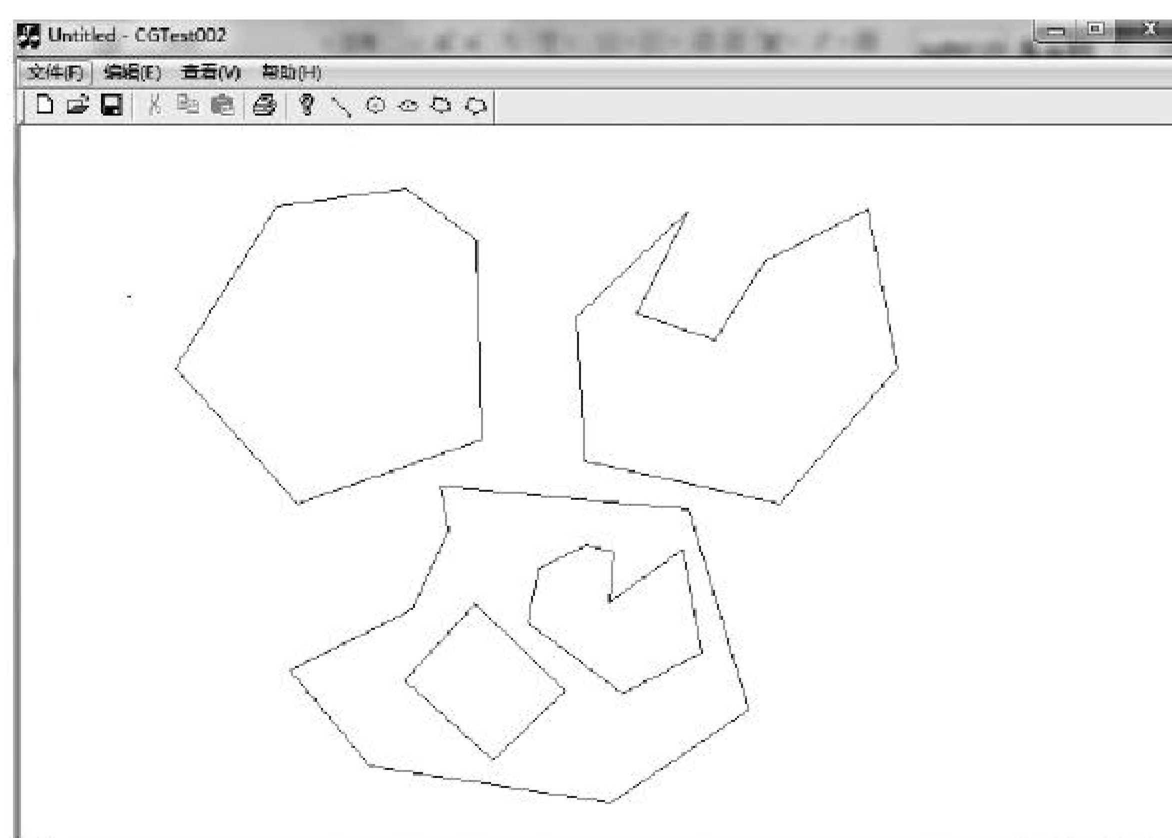


图 3.4-7　多边形

上述的多边形扫描转换算法函数代码参考如下：

```
/******************************************************************
scanTransfer: 多边形扫描转换算法函数
pDC: 显示器指针; polyline: 多边形; crColor: 颜色
******************************************************************/
void scanTransfer(CDC * pDC,CPolyLine &polyline,COLORREF crColor){
    //扫描转换
    int ymin,ymax;
    CArray<CLine,CLine> m_line_Array_Out;
    m_line_Array_Out.Append(polyline.m_PolyLine_array_Out);    //外环多边形
    //得到最小和最大扫描线
    if(m_line_Array_Out.GetAt(0).pt1.y<=m_line_Array_Out.GetAt(0).pt2.y)  {
        ymin=m_line_Array_Out.GetAt(0).pt1.y;
        ymax=m_line_Array_Out.GetAt(0).pt2.y;
    }
    else  {
        ymin=m_line_Array_Out.GetAt(0).pt2.y;
        ymax=m_line_Array_Out.GetAt(0).pt1.y;
    }
    for(int i=1;i<m_line_Array_Out.GetSize();i++)  {
        if(m_line_Array_Out.GetAt(i).pt2.y<ymin)
            ymin=m_line_Array_Out.GetAt(i).pt2.y;
        else if(m_line_Array_Out.GetAt(i).pt2.y>ymax)
            ymax=m_line_Array_Out.GetAt(i).pt2.y;
    }
    //从 ymin 到 ymax 扫描转换
    CArray<int,int> m_x_Array;
    int m_x;                                          //交点
    int j,k;
    for(int yi=ymin;yi<=ymax;yi++)  {
        m_x_Array.RemoveAll();
        //判断扫描线和哪些边相交,如相交,求交点,并排序
        //首先判断扫描线和外环多边形的交点
        for(i=0;i<m_line_Array_Out.GetSize();i++)
```

```
        {//将每条边的最大 y 值缩短一个单位,判断是否和扫描线相交,如相交,求交点,插入交点
集并排序
if((yi>=m_line_Array_Out.GetAt(i).pt1.y&&yi<m_line_Array_Out.GetAt(i).pt2.y)||(yi>=m_
line_Array_Out.GetAt(i).pt2.y&&yi<m_line_Array_Out.GetAt(i).pt1.y))  {//求交点
m_x=GetInterPtXForScanY(yi,m_line_Array_Out.GetAt(i).pt1.x,m_line_Array_Out.GetAt(i).
pt1.y,m_line_Array_Out.GetAt(i).pt2.x,m_line_Array_Out.GetAt(i).pt2.y);
                    OrderToInsertPt_x(m_x_Array,m_x);        //排序
                }
                else if(yi==m_line_Array_Out.GetAt(i).pt1.y&&yi==m_line_Array_Out.GetAt(i).
pt2.y)  {                                                   //是水平线,则将两个端点加入点集
                    OrderToInsertPt_x(m_x_Array,m_line_Array_Out.GetAt(i).pt1.x);
                    OrderToInsertPt_x(m_x_Array,m_line_Array_Out.GetAt(i).pt2.x);
                }
            }
            //再判断扫描线和内环多边形的交点
            CArray<CLine,CLine> m_line_Array_in;
            for(k=0;k<polyline.in_num;k++)  {
                m_line_Array_in.Append(polyline.m_PolyLine_array_in[k]);  //内环多边形
                for(i=0;i<m_line_Array_in.GetSize();i++){//将每条边的最大 y 值缩短一个单
位,判断是否和扫描线相交,如相交,求交点,插入交点集并排序
if((yi>=m_line_Array_in.GetAt(i).pt1.y&&yi<m_line_Array_in.GetAt(i).pt2.y)||(yi>=m_
line_Array_in.GetAt(i).pt2.y&&yi<m_line_Array_in.GetAt(i).pt1.y)){  //求交点
m_x=GetInterPtXForScanY(yi,m_line_Array_in.GetAt(i).pt1.x,m_line_Array_in.GetAt(i).pt1.
y,m_line_Array_in.GetAt(i).pt2.x,m_line_Array_in.GetAt(i).pt2.y);
                        OrderToInsertPt_x(m_x_Array,m_x);  //排序
                    }
                    else if(yi==m_line_Array_in.GetAt(i).pt1.y&&yi==m_line_Array_in.GetAt
(i).pt2.y){                                                 //是水平线,则将两个端点加入点集
                        OrderToInsertPt_x(m_x_Array,m_line_Array_in.GetAt(i).pt1.x);
                        OrderToInsertPt_x(m_x_Array,m_line_Array_in.GetAt(i).pt2.x);
                    }
                }
                m_line_Array_in.RemoveAll();
            }
            for(j=0;j<=m_x_Array.GetSize()-2;j++,j++)  {  //区间对填充色
                for(k=m_x_Array.GetAt(j);k<=m_x_Array.GetAt(j+1);k++)
                pDC->SetPixel(k,yi,crColor);  }
        }
}
```

在上述的扫描线多边形填充算法中,需要求扫描线和多边形边的交点,并对交点进行排序,其中扫描线和多边形边的交点即为水平线和线段的交点,函数代码如下:

```
/* yx:扫描线; x0 y0: 线段一个端点; x1 y1: 线段另一个端点; 返回值为交点 x 值 */
int GetInterPtXForScanY(int yx,int x0,int y0,int x1,int y1){  //求扫描线和线段交点
    int m_x;
    if(yx==y0)
        m_x=x0;
    else if(yx==y1)
        m_x=x1;
    else  {
```

```
        int a = y0 - y1;
        int b = x1 - x0;
        int c = x0 * y1 - x1 * y0;
        double m_dblX;
        m_dblX = ( - 1) * (double(b * yx + c))/(double)a;
        m_x = (int)(m_dblX + 0.5);
    }
    return  m_x;
}
```

交点 x 值排序函数代码如下：

```
/ * m_x_Array: x 值集合; m_x: 待排序 x 值 * /
void OrderToInsertPt_x(CArray< int,int > & m_x_Array,int m_x){  //排序
    for(int i = 0;i < m_x_Array.GetSize();i++)  {
        if(m_x < m_x_Array.GetAt(i))  {
            //插入中间
            m_x_Array.InsertAt(i,m_x);
            return;}
    }
    m_x_Array.Add(m_x);                                 //交点值大,则加入尾部
}
```

利用上述算法对图 3.4-7 所示的多边形进行扫描转换,效果如图 3.4-8 所示。

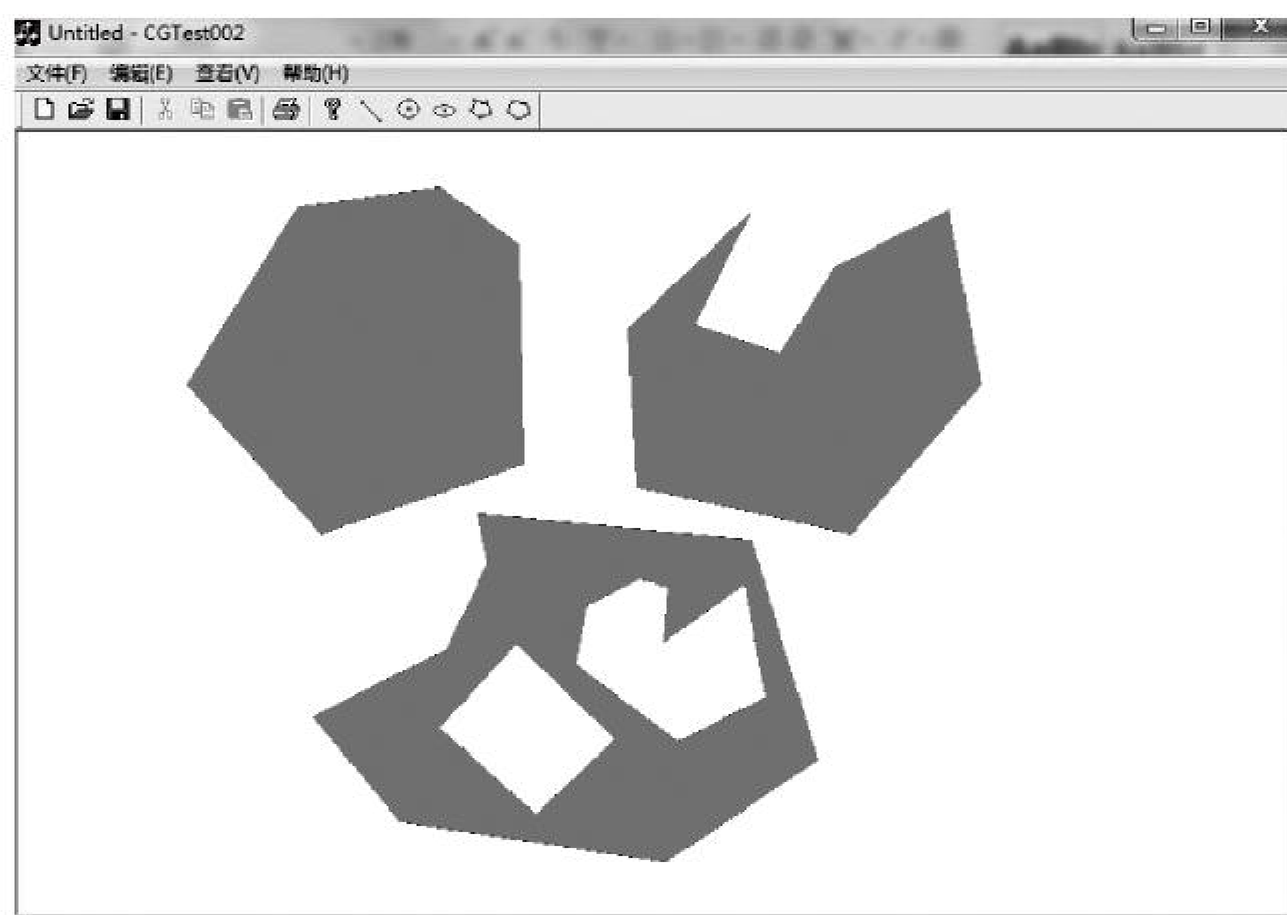


图 3.4-8　多边形扫描转换

由于扫描线填充算法的时间主要花费在计算扫描线与多边形各边的求交运算上,因此提高多边形填充效率的方法是减少与多边形求交点的边数和降低求交工作量,在处理一条扫描线时,仅对和当前扫描线相交的有效边上点组成的区间对进行填充。为此,可以通过构造边表来表示和扫描线对应的有效边。方法如下。

(1) 构造一个多边形的边表(ET),这个边表按照每个边的最小 y 值的增量进行排序。如果边的最小 y 值相等,则按照 x 的增量排序,如果 x 值也相等,则按照每条边的直线斜率 k 的倒数的增量排序。

(2) 当沿着扫描线 y_{scan} 进行扫描时,从边表中顺序取出最小 y 值和扫描线 y 相等的边,

组成有效边表(AET),并对有效边表按照(1)的方式进行排序。然后按照 x 的增量顺序组成两两区间对,并对区间进行填充。

(3) 每一个有效边的 x 值修改为 $x=x+\frac{1}{k}$,最小 y 值修改为 $y=y_{\min}=y_{\min}+1$。

(4) 令扫描线 $y_{scan}=y_{scan}+1$,然后判断有效边表中每个边的最大 y 值,如果 $y_{\max}<y_{scan}$,说明该边不再和扫描线相交,则将该边从有效边表中删除,然后,转入(2)。

(5) 当有效边表中有效边的数量为空的时候,说明已经处理到多边形最大 y 值,扫描转换结束。

上述的有效边表扫描线算法,对于每一个扫描线,只处理扫描线通过的边,在求交点时,是利用递推公式 $x=x+\frac{1}{k}$ 计算的,因此扫描效率更高。

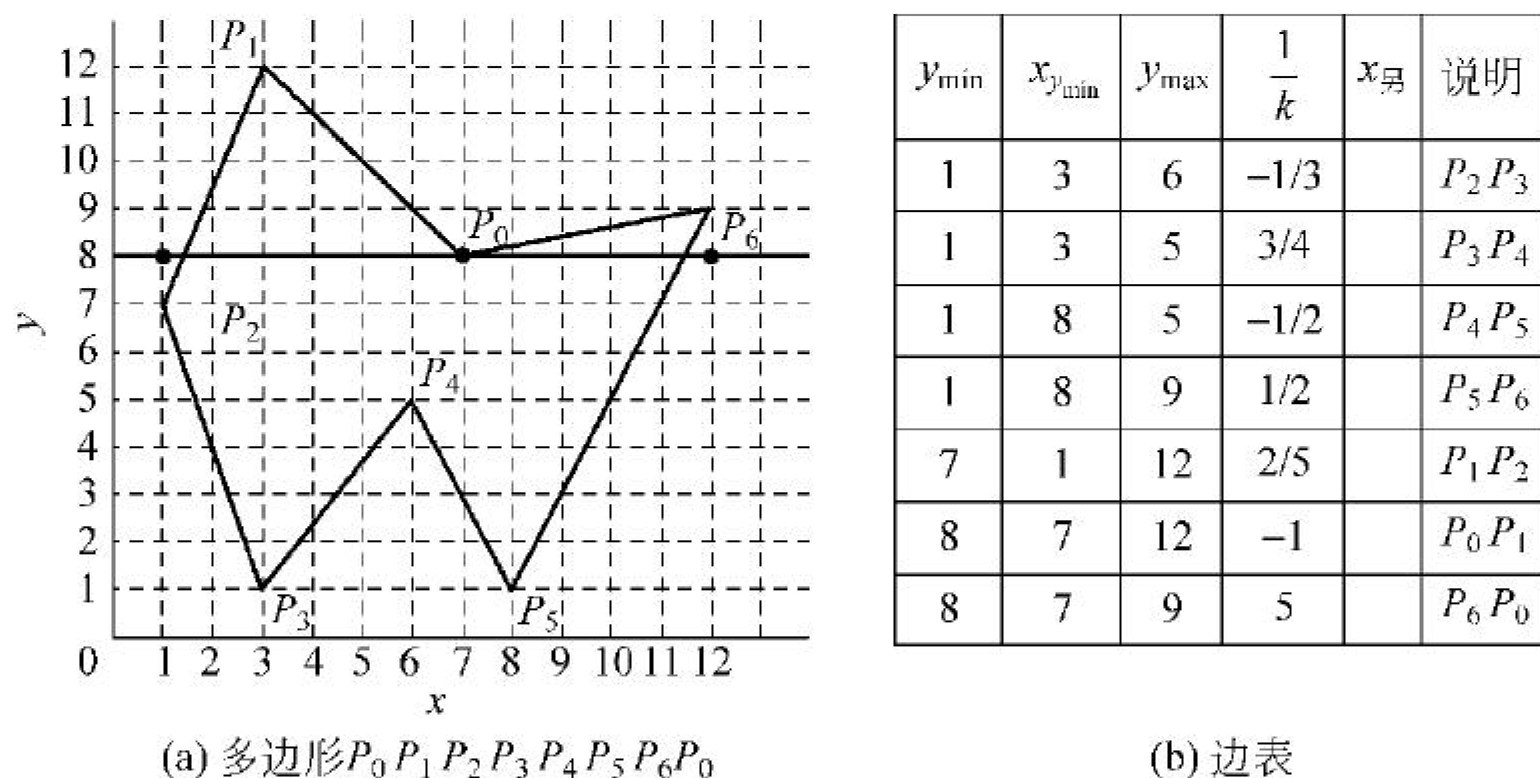
在构造每条边的信息时,需要包括:该边当前最小 y 值、当前对应的 x 值、该边最大 y 值以及该边斜率的倒数。当该边为特殊位置直线水平线时,斜率倒数为无穷大,为了准确记录边的信息,此时还应该记录该边另外一个顶点的 x 值信息;如果斜率倒数不为无穷大,则另一个顶点的 x 值可不必记录。因此,构造边结构如下:

$y_{\min}$	$x_{y_{\min}}$	$y_{\max}$	$\frac{1}{k}$	$x_{另}$

对应的边类为:

```
class CAET{                              //构造边类
public:
    int y_min;                           //当前边的最小 y 值
    double x;                            //当前边的 x 值
    int y_max;                           //当前边的最大 y 值
    double k;                            //斜率倒数 k = dy/dx
    int x1;                              //当斜率为无穷大时记录另外一个点的 x 值
};
```

图 3.4-9 所示为一多边形以及根据上述方法构造的边表。



$y_{\min}$	$x_{y_{\min}}$	$y_{\max}$	$\frac{1}{k}$	$x_{另}$	说明
1	3	6	–1/3		P_2P_3
1	3	5	3/4		P_3P_4
1	8	5	–1/2		P_4P_5
1	8	9	1/2		P_5P_6
7	1	12	2/5		P_1P_2
8	7	12	–1		P_0P_1
8	7	9	5		P_6P_0

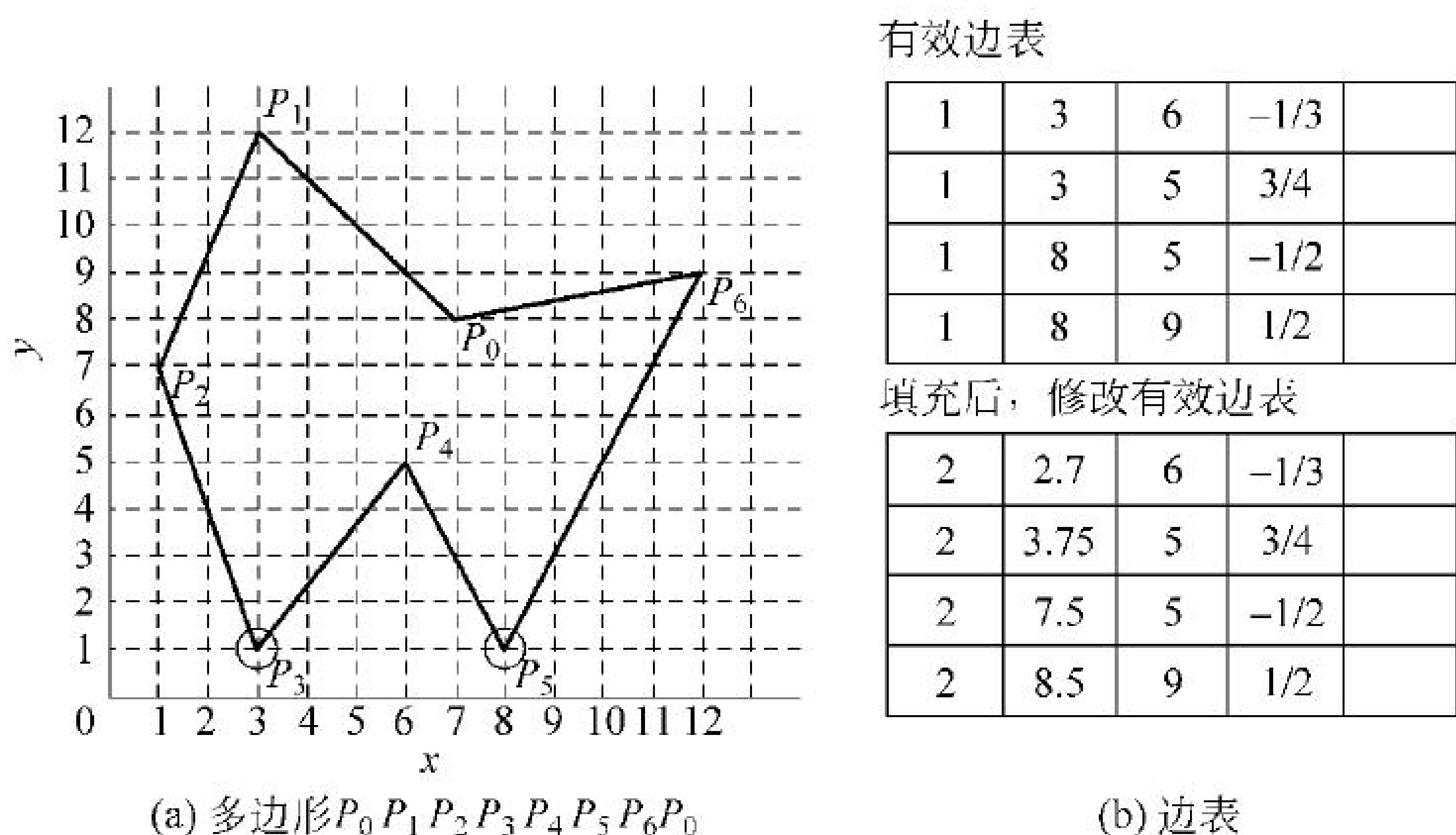
(a) 多边形 $P_0P_1P_2P_3P_4P_5P_6P_0$　　(b) 边表

图 3.4-9　多边形及边表

在边表中，由于斜率倒数没有无穷大的情况，因此边表中的 $x_{另}$ 这一列不用登记。其中，在构造直线 P_2P_3 的节点时，在最大顶点即(1,7)处，由于共享该顶点的多边形另外一条边即 P_1P_2 分布在扫描线 $y=7$ 的另外一侧，考虑交点的取舍为题，将该直线的 y_{max} 减少一个单位，即节点值为边表值的第一行：

1	3	6	−1/3	

当扫描线 $y=1$ 时，从边表中取出有效边，通过排序组成有效边表，将有效边表中的边的 x 值两两构成区间对填充，然后修改有效边的值：$x=x+\frac{1}{k}$，$y_{min}=y_{min}+1$，如图 3.4-10 所示。如果 $y_{max}<y_{min}$，则该边已经处理完，不再为有效边，将其从有效边表中删除。



有效边表

1	3	6	−1/3	
1	3	5	3/4	
1	8	5	−1/2	
1	8	9	1/2	

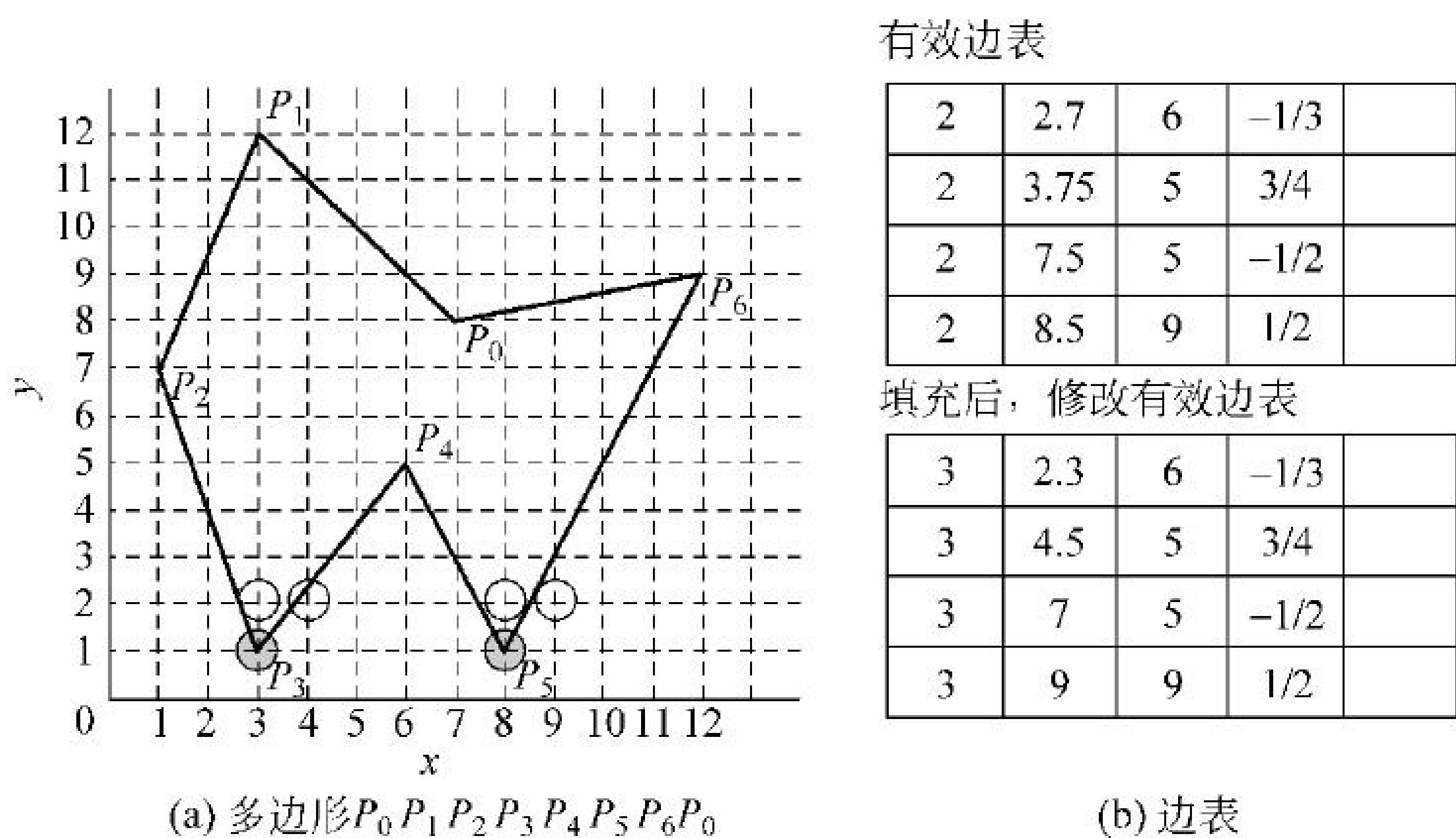
填充后，修改有效边表

2	2.7	6	−1/3	
2	3.75	5	3/4	
2	7.5	5	−1/2	
2	8.5	9	1/2	

(a) 多边形 $P_0P_1P_2P_3P_4P_5P_6P_0$　　(b) 边表

图 3.4-10　扫描线 $y=1$

以此类推，当扫描线 $y=2$ 时，首先判断边表中是否存在有效边。如有，则取出插入现有的有效边表中，并排序，同样将有效边表中的边的 x 值两两构成区间对，填充，然后修改有效边的值：$x=x+\frac{1}{k}$，$y_{min}=y_{min}+1$，判断是否 $y_{max}<y_{min}$ 的情况，如图 3.4-11 所示。



有效边表

2	2.7	6	−1/3	
2	3.75	5	3/4	
2	7.5	5	−1/2	
2	8.5	9	1/2	

填充后，修改有效边表

3	2.3	6	−1/3	
3	4.5	5	3/4	
3	7	5	−1/2	
3	9	9	1/2	

(a) 多边形 $P_0P_1P_2P_3P_4P_5P_6P_0$　　(b) 边表

图 3.4-11　扫描线 $y=2$

循环执行上述操作，当 $y_{max}<y_{min}$ 时，从有效边表中删除已经处理完的边，例如，$y=5$ 时，将 P_3P_4 以及 P_4P_5 两个边从有效边表中删除，当 $y=7$ 时，将 P_1P_2 边从边表中取出，插入有效边表并排序，开始填充，并重复上述的修改边的值以及判断操作。当边表中的所有边都已经取出，而且有效边表也成为空表，说明所有边都已经处理，此时多边形内部全部填充完毕，扫描转换结束，如图 3.4-12 所示。

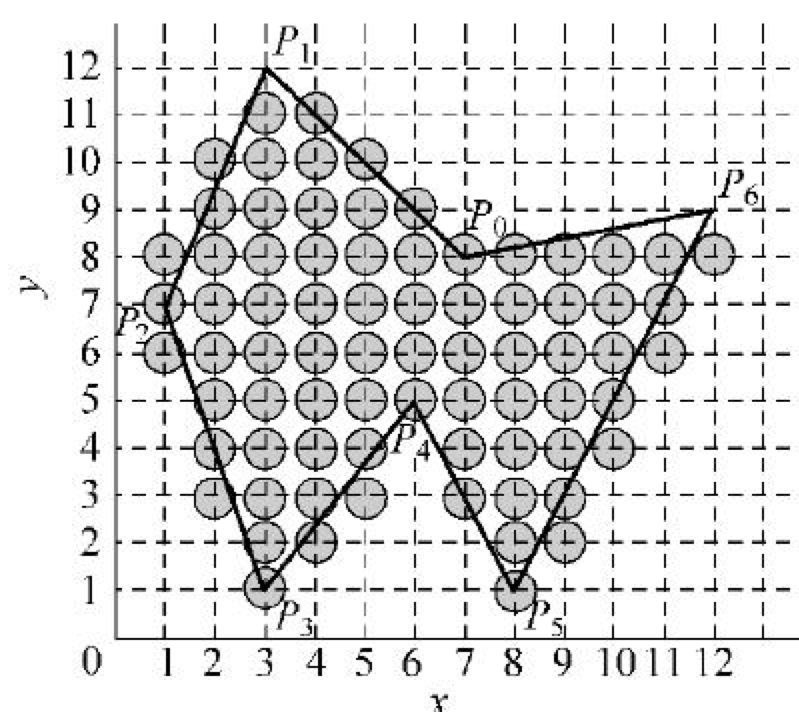


图 3.4-12 多边形填充

上述的有效边表多边形扫描转换算法函数代码参考如下：

```
/*************************************************************
scanTransfer_AET: 有效边表算法; polyline: 多边形; crColor: 填充色
*************************************************************/
void scanTransfer_AET(CDC * pDC,CPolyLine &polyline,COLORREF crColor)
{//有效边表扫描转换
    //对多边形所有的边构造边表
    CArray< CAET,CAET > ET_array;
    //处理外环边
    CArray< CLine,CLine > m_line_Array_Out;
    m_line_Array_Out.Append(polyline.m_PolyLine_array_Out);       //外环多边形
    for(int i = 0;i < m_line_Array_Out.GetSize();i++)             //构造边,并插入链表中
        InsertAndSortChain(ET_array,BuildAET(m_line_Array_Out.GetAt(i)));
    //处理内环边
    CArray< CLine,CLine > m_line_Array_in;
    for(int k = 0;k < polyline.in_num;k++){
        m_line_Array_in.Append(polyline.m_PolyLine_array_in[k]); //内环多边形
        for(i = 0;i < m_line_Array_in.GetSize();i++)             //构造边,并插入链表中
            InsertAndSortChain(ET_array,BuildAET(m_line_Array_in.GetAt(i)));
        m_line_Array_in.RemoveAll();
    }
    //构造有效边表
    CArray< CAET,CAET > m_AET_array;
    int y_scan;                                                  //扫描线
    CAET aet_tmp;
    //先取出第一个节点
    aet_tmp = ET_array.GetAt(0);
    y_scan = aet_tmp.y_min;
    ET_array.RemoveAt(0);                                        //从边表中移出该边
    InsertAndSortChain(m_AET_array,aet_tmp);                     //插入有效边
    while(1){
        while(1)  {//从边表中取出当前的边,插入有效边并排序
            if(ET_array.GetSize()> 0){
                aet_tmp = ET_array.GetAt(0);
                if(aet_tmp.k == MAX_VALUE){
                    //特殊情况,直接填充
                    int x0;
                    if(aet_tmp.x < aet_tmp.x1) x0 = aet_tmp.x; else x0 = aet_tmp.x1;
```

```
                    for(k = x0;k <= abs(aet_tmp.x - aet_tmp.x1);k++)
                        pDC -> SetPixel(k,y_scan,crColor);
                    ET_array.RemoveAt(0);                    //从边表中移出该边
                }
                else if(y_scan == aet_tmp.y_min){
                    InsertAndSortChain(m_AET_array,aet_tmp);
                    ET_array.RemoveAt(0);                    //从边表中移出该边
                }
                else
                    break;
            }
            else
                break;
        }
        //两两区间配对填充
        for(i = 0;i < m_AET_array.GetSize();i++,i++){
for(k = (int)(m_AET_array.GetAt(i).x + 0.5);k <= (int)(m_AET_array.GetAt(i + 1).x + 0.5);k++)
                pDC -> SetPixel(k,y_scan,crColor);   }
        y_scan++;                                            //扫描线 + 1
        if(m_AET_array.GetSize()> 0){
            for(i = 0;i < m_AET_array.GetSize();i++)
            {//从边表中取出当前的边
                aet_tmp = m_AET_array.GetAt(i);
                if(aet_tmp.y_max < y_scan){
                    m_AET_array.RemoveAt(i);                 //删除该节点
                    i--;   }
                else  {
                    aet_tmp.y_min++;
                    if(aet_tmp.k!= MAX_VALUE)
                        aet_tmp.x += aet_tmp.k;
                    m_AET_array.InsertAt(i,aet_tmp);
                    m_AET_array.RemoveAt(i + 1);             //删除该节点
                }
            }
        }
        else
            break;                                       //已经没有有效边,则完成扫描
    }
}
```

函数中在构造多边形边表和有效边表时,都需要对插入边进行排序,函数代码如下:

```
/* AET_array: 已排序的有效边集合; aet: 待排序的有效边 */
void InsertAndSortChain(CArray< CAET,CAET> &AET_array,CAET &aet)
{//将构造的有效边插入边表中,并根据 ymin、x,以及 k 排序
    if(AET_array.GetSize() == 0)
        AET_array.Add(aet);
    else  {
        for(int i = 0;i < AET_array.GetSize();i++)  {
            //调出每一个边,进行排序比较
            CAET tmpAET;
```

```
                tmpAET = AET_array.GetAt(i);
                if(aet.y_min < tmpAET.y_min){          //比现在的小,则将边插入其前边,并返回
                    AET_array.InsertAt(i,aet);
                    return;
                }
                else if(aet.y_min == tmpAET.y_min){    //再判断 x 值
                    if(aet.x < tmpAET.x)   {           //比现在的 x 小,则插入其前边,并返回
                        AET_array.InsertAt(i,aet);
                        return;
                    }
                    else if(aet.x == tmpAET.x){        //如果相等,则判断斜率 k
                        if(aet.k <= tmpAET.k){         //如果比现在的 k 小,则插入其前边,并返回
                            AET_array.InsertAt(i,aet);
                            return;
                    }
                    }
                }
            }
            //循环后,仍没有插入,则插入最后边
            AET_array.Add(aet);
            return ;
        }
    }
```

执行应用程序即可实现填充。在应用程序中,可以设计一个非模式对话框,来选择不同的扫描转换算法,实现多边形填充,效果如图 3.4-13 所示。

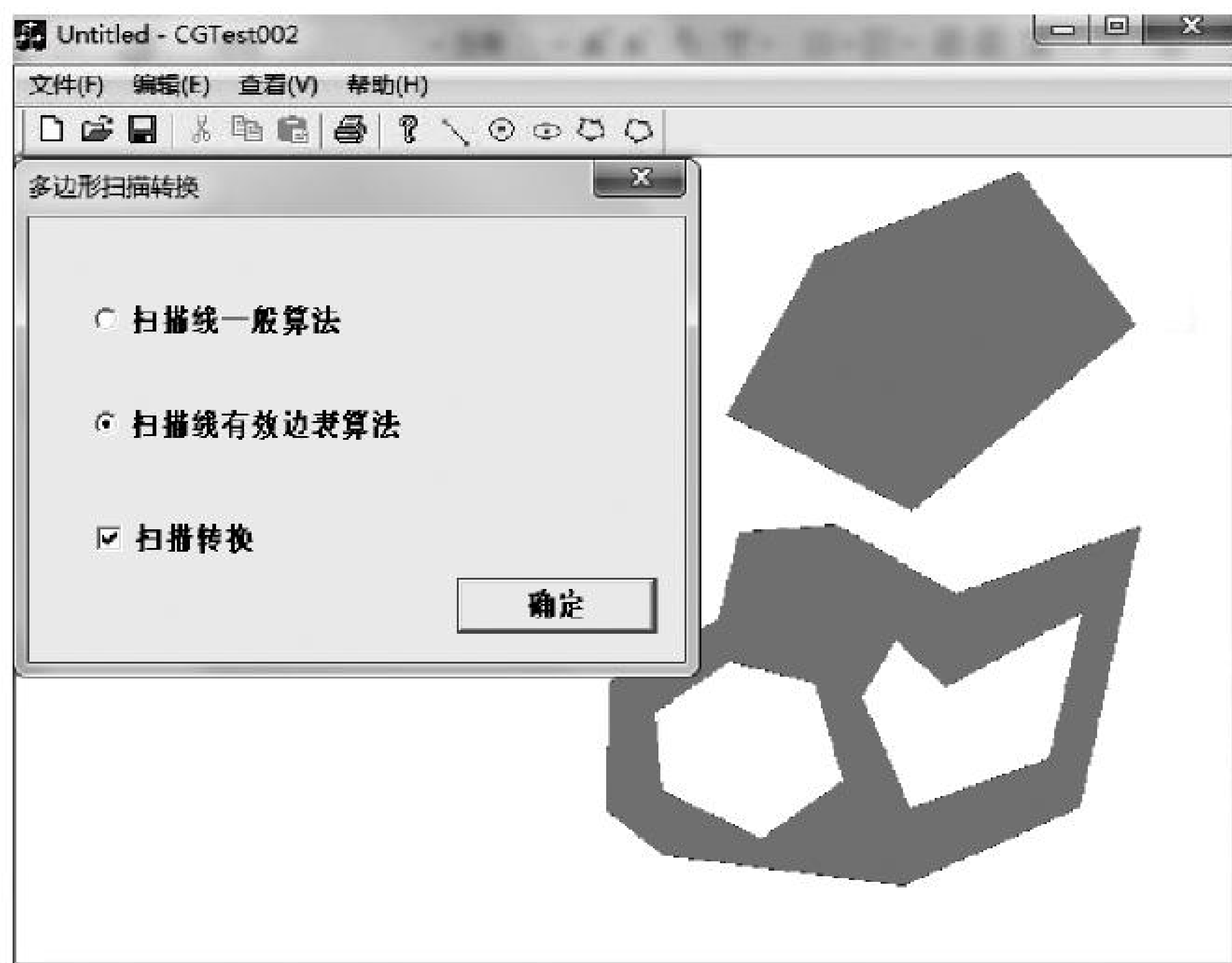


图 3.4-13 多边形扫描转换实现

除了上述的扫描线算法外,多边形扫描转换算法还有边缘填充法、栅栏填充法以及边界标志法等。

边缘填充算法的基本思想:按任意顺序处理多边形的每条边。处理时,先求出该边与扫描线的交点,再对扫描线上交点右方的所有像素颜色取反。边缘填充算法思路简单,但对

于复杂图形，每一像素可能被访问多次。

栅栏填充算法是借助栅栏的原理来实现的，栅栏指的是一条通过多边形顶点且与扫描线垂直的直线，它把多边形分为两半。算法基本思想：按任意顺序处理多边形的每一条边，当处理每条边与扫描线的交点时，将交点与栅栏之间的像素取反。这种算法尽管减少了被重复访问像素的数目，但仍有一些像素被重复访问。

边界标志算法的基本思想：先用特殊的颜色在帧缓存中将多边形的边界勾画出来，然后将着色的像素点依 x 坐标递增的顺序配对，再把每一对像素构成的区间置为填充色。边界标志算法分为两个步骤：打标记；填充。当用软件实现本算法时，速度与改进的有效边表算法相当，但本算法用硬件实现后速度会有很大提高。

限于篇幅，本书对这些算法不再详述。

3.4.2 区域填充

这里的区域指已经表示成点阵形式的填充图形，它是像素的集合。区域可采用内点表示和边界表示两种表示形式。区域填充指先将区域的一点赋予指定的颜色，然后将该颜色扩展到整个区域的过程。

区域填充分为两类：多边形的扫描转换与种子填充。两者的前提条件不同，前者需提供多边形各顶点的坐标及填充色或图案；后者则要求给出边界颜色特征及区域内的一个点（称为种子点）的坐标。多边形的扫描转换主要是通过确定穿越区域的扫描线的覆盖区间来填充，种子填充是从给定的位置开始涂描直到指定的边界条件为止。

区域填充要求区域是连通的。区域连通有四向连通区域和八向连通区域两种，如图 3.4-14 所示。四向连通区域指的是从区域上一点出发，通过上、下、左、右移动，即可到达区域内的任意像素；八向连通区域除了四向连通区域的四个方向外，还需要通过左上、右上、左下、右下这四个方向，才能到达区域内的任意位置。

一般来说，八向连通算法可以填充四向连通区域，而四向连通算法有时不能填充八向连通区域。例如，八向连通填充算法能够正确填充图 3.4-15 所示的区域内部，而四向连通填充算法只能完成该区域的部分填充。

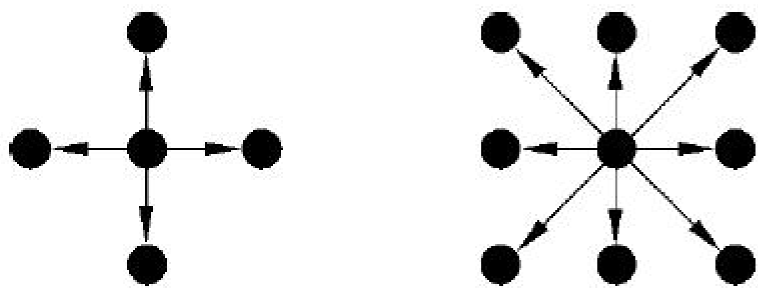


图 3.4-14 四向连通和八向连通

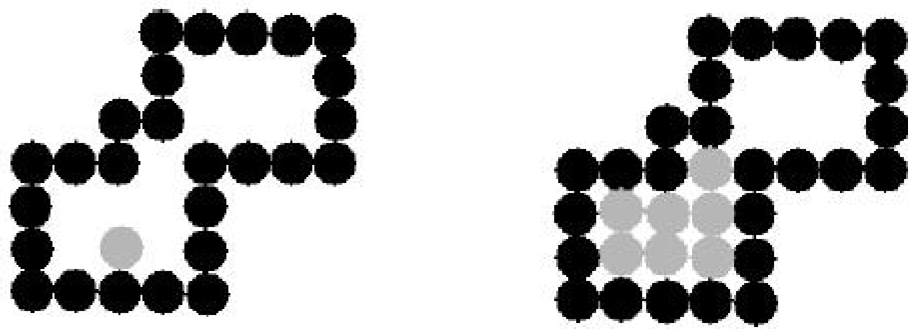


图 3.4-15 连通区域

在编程时，上面的连通算法可以通过递归函数的方式实现。例如八向连通算法的递归函数的代码参考如下：

```
/************************************************************************
FloodFill8: 八向连通填充算法函数
pDC: 显示器设备指针; x,y: 种子点; oldColor: 区域原来的颜色; newColor: 新填充色
************************************************************************/
```

```
void FloodFill8(CDC * pDC, int x, int y, COLORREF oldColor, COLORREF newColor){
    if(pDC->GetPixel(CPoint(x,y)) == oldColor){
        pDC->SetPixel(x,y,newColor);
        FloodFill8(pDC,x,y+1,oldColor,newColor);
        FloodFill8(pDC,x,y-1,oldColor,newColor);
        FloodFill8(pDC,x-1,y,oldColor,newColor);
        FloodFill8(pDC,x+1,y,oldColor,newColor);
        FloodFill8(pDC,x+1,y+1,oldColor,newColor);
        FloodFill8(pDC,x+1,y-1,oldColor,newColor);
        FloodFill8(pDC,x-1,y+1,oldColor,newColor);
        FloodFill8(pDC,x-1,y-1,oldColor,newColor);
    }
}
```

上述递归算法简单，但是效率不高，区域内每一个像素都引起一次递归，有些像素会多次重复递归判断。这样不但降低了算法效率，而且占用了很大的内存空间，费时费内存，在实际实现时，当图像相对比较复杂时会存在内存溢出的现象。

递归算法的一种改进方法是扫描线种子填充算法，其基本过程如下：当给定种子点(x, y)时，首先分别向左和向右两个方向填充种子点所在扫描线上的位于给定区域的一个区段，同时记下这个区段的范围[xLeft, xRight]，然后确定与这一区段相连通的上、下两条扫描线上位于给定区域内的区段，并依次保存下来。反复进行这个过程，直到填充结束。

扫描线种子填充算法可由下列四个步骤实现。

(1) 初始化一个空的栈用于存放种子点，将种子点(x, y)入栈。

(2) 判断栈是否为空，如果栈为空则结束算法，否则取出栈顶元素作为当前扫描线的种子点(x, y)，y 是当前的扫描线。

(3) 从种子点(x, y)出发，沿当前扫描线向左、右两个方向填充，直到边界。分别标记区段的左、右端点坐标为 xLeft 和 xRight。

(4) 分别检查与当前扫描线相邻的 y−1 和 y+1 两条扫描线在区间[xLeft, xRight]中的像素，从 xLeft 开始向 xRight 方向搜索，若存在非边界且未填充的像素点，则找出这些相邻的像素点中最右边的一个，并将其作为种子点压入栈中，然后返回第(2)步。

上述扫描线种子填充算法，可以实现在已知区域的边界像素颜色或者已知区域内部的颜色这两种情况下，将区域内部用新的颜色填充。下面是已知区域内部颜色的填充函数参考代码，对于已知区域边界像素颜色的情况，只需将函数中判断区域内部颜色的代码改成判断区域边界颜色的代码即可。

```
/*************************************************************************
ScanLineSeedFill:扫描线种子填充函数
pDC: 显示器设备指针; x,y: 种子点; oldColor: 区域原来的颜色; newColor: 新填充色
*************************************************************************/
void ScanLineSeedFill(CDC * pDC, int x, int y, COLORREF oldColor, COLORREF newColor){
    CArray<CPoint,CPoint> m_stk_point;                  //构造种子点栈
    m_stk_point.Add(CPoint(x,y));                       //第(1)步,种子点入栈
    CPoint seedPt;
    int xl,xry1;
    while(m_stk_point.GetSize()>0)  {
```

```
        seedPt = m_stk_point.GetAt(0);                    //第(2)步,取当前种子点
        m_stk_point.RemoveAt(0);
        x = seedPt.x;
        y = seedPt.y;
        xl = xr = x;
        yl = y;
         //第(3)步,向左右填充(在当前点所在扫描线扫描)
        if(pDC->GetPixel(seedPt) == oldColor){
            pDC->SetPixel(x,y,newColor);                  //填充该点
            Fill(pDC,xl,yl,-1,oldColor,newColor);         //x--方向填充,并返回边界点
            Fill(pDC,xr,yl,1,oldColor,newColor);          //x++方向填充,并返回边界点
            //第(4)步,处理相邻两条扫描线,并获得新种子入栈
            SearchLineNewSeed(pDC,m_stk_point,xl,xr,y-1,oldColor,newColor);
            SearchLineNewSeed(pDC,m_stk_point,xl,xr,y+1,oldColor,newColor);}
    }
}
```

其中,在一条扫描线上填充当前区段的函数代码如下:

```
/* pDC: 显示器指针; x,y: 当前扫描线初始点位置; drFlg: 填充方向; oldColor: 区域原来颜色;
newColor: 填充色 */
void Fill(CDC *pDC,int &x,int &y,int drFlg,COLORREF oldColor,COLORREF newColor){
    if(drFlg==-1){//x--
        for(;pDC->GetPixel(CPoint(x-1,y)) == oldColor;){
            pDC->SetPixel(--x,y,newColor);                //填充该点
        }
    }
    else  {//x++
        for(;pDC->GetPixel(CPoint(x+1,y)) == oldColor;){
            pDC->SetPixel(++x,y,newColor);                //填充该点
        }
    }
}
```

处理相邻两条扫描线,并获得新种子入栈的函数代码如下:

```
/* m_stk_point: 入栈的种子点; xLeft, xRight: 当前扫描线边界; y:当前扫描线 */
void SearchLineNewSeed(CDC *pDC,CArray<CPoint,CPoint> &m_stk_point, int xLeft, int xRight,
int y, COLORREF oldColor,COLORREF newColor){
    int xt = xLeft;
    bool findNewSeed = false;
    while(xt <= xRight)  {
        //从 xl 开始到 xr,找到新的种子点
        if(pDC->GetPixel(CPoint(xt,y)) == oldColor)  {
            findNewSeed = true;                           //说明会有种子点
            xt++;
            continue;}
        else  {
            if(findNewSeed == true){
                //当前点不是,则前一点是种子点
                m_stk_point.Add(CPoint(xt-1,y));
                findNewSeed = false;
```

```
                    xt++;
                    continue;
                }
                else  {
                    //没有找到,则继续
                    xt++;
                    continue;
                }
            }
        }
        if(findNewSeed == true){
            m_stk_point.Add(CPoint(xRight,y));              //把右边界作为种子加入
        }
    }
```

3.5 字符和汉字的表示

在计算机图形学中,字符指计算机在文本方式下能够在屏幕上显示的数字、字母、音标、标点符号、数学符号、汉字等符号。计算机中的字符由一个数字编码唯一标识。最流行的字符集是美国信息交换用标准代码集,简称 ASCII 码。它用 7 位二进制编码规定了 129 个字符代码,其中代码 0~31 表示控制字符,32~127 表示字母、标点符号、数学符号以及一些特殊符号。

我国除采用 ASCII 码外,还另外制定了汉字编码的国家标准字符集,如:《信息交换用汉字编码字符集基本集》(GB 2312—1980)。该字符集分为 94 个区,94 个位,每个符号由一个区码和一个位码共同标识。区码和位码各用一个字节表示。为了能够区分 ASCII 码与汉字编码,采用字节的最高位来标识:最高位为 0 表示 ASCII 码;最高位为 1 表示汉字编码。共收录了 6763 个常用汉字。2000 年 3 月信息产业部和国家质量技术监督局又颁布了国家标准《信息交换用汉字编码字符集基本集的扩充》(GB 18030—2000)。它共收录了 2.7 万多个汉字,总编码空间超过 150 万个码位,采用单/双/四字节混合编码,与现有绝大多数操作系统、中文平台在内码一级兼容,可支持现有应用系统,并包容了其中收录的所有汉字和蒙、藏、彝、维等少数民族文字。

为了在显示器等输出设备上输出字符,计算机系统中必须安装相应的字库。字库分为点阵字库和矢量字库两种,用于存储每个字符的形状信息。点阵字库中,每个字符用二值点阵信息表示每个字符,矢量字库则用直线和曲线(如三次 B 样条曲线/Bézier 曲线)来描述每个字符的轮廓形状。

1. 点阵字符

在点阵字库中,每个字符由一个位图表示(如图 3.5-1 所示),并把它用一个称为字符掩膜的矩阵来表示,其中的每个元素都是一位二进制数。如果该位为 1,表示字符的笔画经过此位,该像素置为字符颜色;如果该位为 0,表示字符的笔画不经过此位,该像素置为背景颜色。点阵字符的显示分为两步:首先从字库中将它的位图检索出来,然后将检索到的位图写到帧缓存器中。

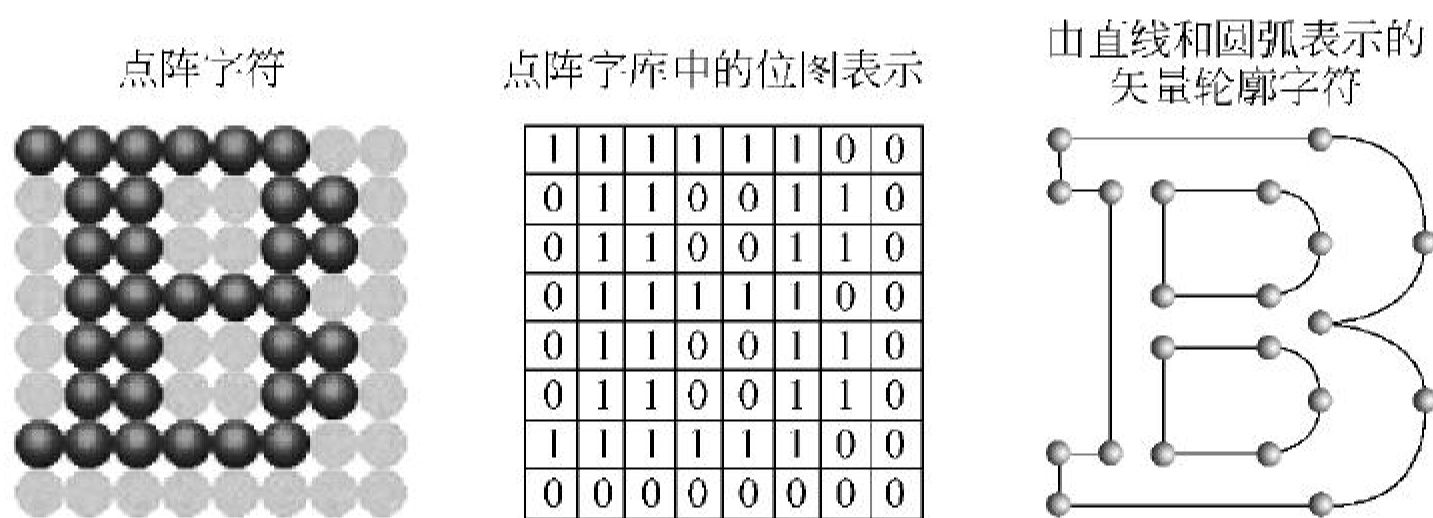


图 3.5-1 字符的点阵表示和矢量轮廓表示

在实际应用中,同一个字符有多种字体(如宋体、楷体等),每种字体又有多种大小型号,因此字库的存储空间十分庞大。为了减少存储空间,一般采用压缩技术。

2. 矢量字符

矢量字符记录字符的笔画信息而不是整个位图,具有存储空间小、美观、变换方便等优点。例如:在 AutoCAD 中使用图形实体——形(Shape)来定义矢量字符,其中,采用了直线和圆弧作为基本的笔画来对矢量字符进行描述。

对于字符的旋转、放大、缩小等几何变换,点阵字符需要对其位图中的每个像素进行变换;而矢量字符则只需要对其几何图素进行变换就可以了,例如,对直线笔画的两个端点进行变换,对圆弧的起点、终点、半径和圆心进行变换等。

矢量字符的显示也分为两步:首先从字库中将它的字符信息提取出来;然后取出端点坐标,对其进行适当的几何变换,再根据各端点的标志显示出字符。

轮廓字形法是当今国际上最流行的一种字符表示方法,其压缩比大,且能保证字符质量。轮廓字形法采用直线、B 样条/Bézier 曲线的集合来描述一个字符的轮廓线。轮廓线构成一个或若干个封闭的平面区域。轮廓线定义加上一些指示横宽、竖宽、基点、基线等控制信息就构成了字符的压缩数据。

3.6 线宽和线型处理

3.6.1 线宽处理

线宽是图线除颜色外的另外一个属性,图形光栅化后,图线的最小线宽就是显示设备基本像素点的大小,当采用不同的线宽时,线宽与基本像素点之间呈倍数关系。

常用的线宽处理方法有以下几种。

1. 像素复制方法

像素复制方法又称为线刷子法,在生成具有一定宽度的直线时,可以沿着生成直线时获得的像素点,通过移动一把具有一定宽度的"线刷子"来获得宽度,如图 3.6-1 所示。当直线的斜率在−1～1 之间时,线刷子设定成垂直方向,并将线刷子中心点对准直线上某一像素点,然后将线刷子沿直线运动,就刷出一条具有一定宽度的直线;当直线斜率不在−1～1 之间时,把线刷子设成水平方向,沿着直线运动。

线刷子法的特点是实现简单、效率高，但是获得的直线宽度在倾斜方向和在水平或者垂直方向的宽度不均匀，当线宽为偶数个像素时，线的中心将偏移半个像素；直线的始末端总是水平或者垂直的，不太自然，在两条直线的拐角顶点处会有缺口，如图 3.6-2 所示。在拐角点需要进行连接处理，以消除这个缺口。

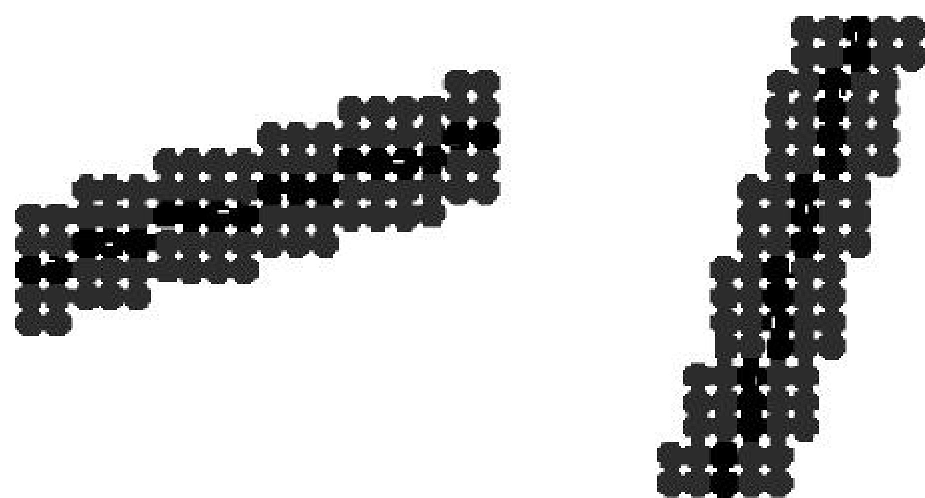


图 3.6-1 线刷子法

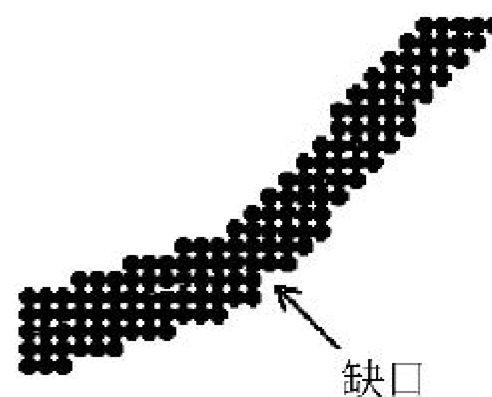


图 3.6-2 两线连接处有缺口

以下是线刷子法绘制某一点的线宽函数的参考代码：

```
/******************************************
LineBrush: 线刷子法绘制某一点的线宽函数
pDC: 显示器指针; x,y: 直线上某点; kFlag: 斜率; 0: 斜率≤1,平行绘制; 1: 斜率>1; 垂直绘制
******************************************/
void LineBrush(CDC * pDC,int x,int y,COLORREF crColor,int kFlag){
    if(kFlag == 0)  {
        pDC->SetPixel(x,y,crColor);
        pDC->SetPixel(x,y-1,crColor);
        pDC->SetPixel(x,y-2,crColor);
        pDC->SetPixel(x,y-3,crColor);
        pDC->SetPixel(x,y+1,crColor);
        pDC->SetPixel(x,y+2,crColor);
        pDC->SetPixel(x,y+3,crColor);
    }
    else  {
        pDC->SetPixel(x,y,crColor);
        pDC->SetPixel(x-1,y,crColor);
        pDC->SetPixel(x-2,y,crColor);
        pDC->SetPixel(x-3,y,crColor);
        pDC->SetPixel(x+1,y,crColor);
        pDC->SetPixel(x+2,y,crColor);
        pDC->SetPixel(x+3,y,crColor);
    }
}
```

在本章前述直线等基本图形扫描转换函数中，在绘制每个像素点时，增加如下代码，判断是否按粗实线绘制：

```
if(lineWidth == 0)//lineWidth 是线宽判定变量, = 0 是单位像素线宽,≠0 是粗实线线宽
    pDC->SetPixel((int)x,(int)(y+0.5),crColor);
else
    LineBrush(pDC,(int)x,(int)(y+0.5),crColor,0);
```

图 3.6-3 是利用上述算法绘制的一定线宽直线效果图。从图中可以看出，不同倾斜角度下线宽不均匀，在两条线段连接处，会存在缺口的现象。

2. 移动刷子方法

移动刷子方法又称方刷子法，它是通过把边长为指定线宽的正方形的中心沿直线作平行移动，来获得具有一定线宽的线条。方刷子法最简单的实现方法是把正方形中心对准单像素宽的直线上的各个像素点，把正方形内的像素全部设置成线条的颜色，效果如图 3.6-4 所示。



图 3.6-3　线刷子法绘制线宽

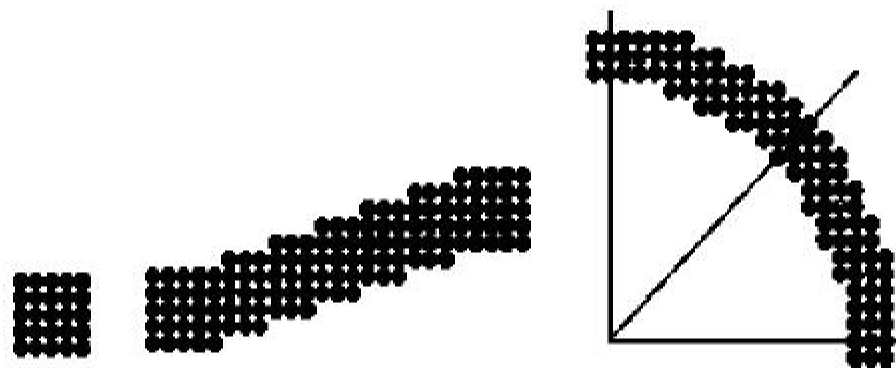


图 3.6-4　方刷子法

方刷子法的特点是简单、容易实现，但是效率低，在线的末端总是水平或者垂直的，在线宽是像素的偶数倍时无法实现，绘制的线条会带有一个“方线帽”，在写像素时，相邻两个像素的方形会重叠，因此会重复写像素。

方刷子法绘制线宽的函数代码参考如下：

```
/*****************************************
SquarBrush: 方刷子法绘制某一点的线宽函数
pDC: 显示器指针; x,y: 直线上某点
*****************************************/
void SquarBrush(CDC *pDC,int x,int y,COLORREF crColor){
    pDC->SetPixel(x,y,crColor);
    pDC->SetPixel(x,y-1,crColor);
    pDC->SetPixel(x,y+1,crColor);
    pDC->SetPixel(x+1,y,crColor);
    pDC->SetPixel(x-1,y,crColor);
    pDC->SetPixel(x-1,y-1,crColor);
    pDC->SetPixel(x-1,y+1,crColor);
    pDC->SetPixel(x+1,y-1,crColor);
    pDC->SetPixel(x+1,y+1,crColor);
}
```

同样，在直线等基本图形扫描转换函数中，在绘制每个像素点时，增加如下代码：

```
if(lineWidth==0)      //lineWidth是线宽判定变量,=0是单
                      //位像素线宽,≠0是粗实线线宽
    pDC->SetPixel((int)x,(int)(y+0.5),crColor);
else
    SquarBrush(pDC,(int)x,(int)(y+0.5),crColor,0);
```

图 3.6-5 是利用上述方刷子法代码实现的线宽处理效果。



图 3.6-5　方刷子法线宽处理

3. 填充图元方法

填充图元法的原理是利用等距和平行线先画出线段内

外边界，然后在内外边界之间进行填充，从而获得指定线宽的图形。这种方法的优点是生成的图形质量高；缺点是计算量大，有些图形的等距线不容易求(例如椭圆弧)。

3.6.2 线型处理

在工程上绘制产品图形时，有时需要根据形状表达的要求，采用不同的线型绘制，如实线、虚线、点划线、双点划线等，因此，线型也是图线的另一个重要属性。除了实线外，其他线型都是不连续的线型，所以，需要考虑如何表示图线中不连续的部分。

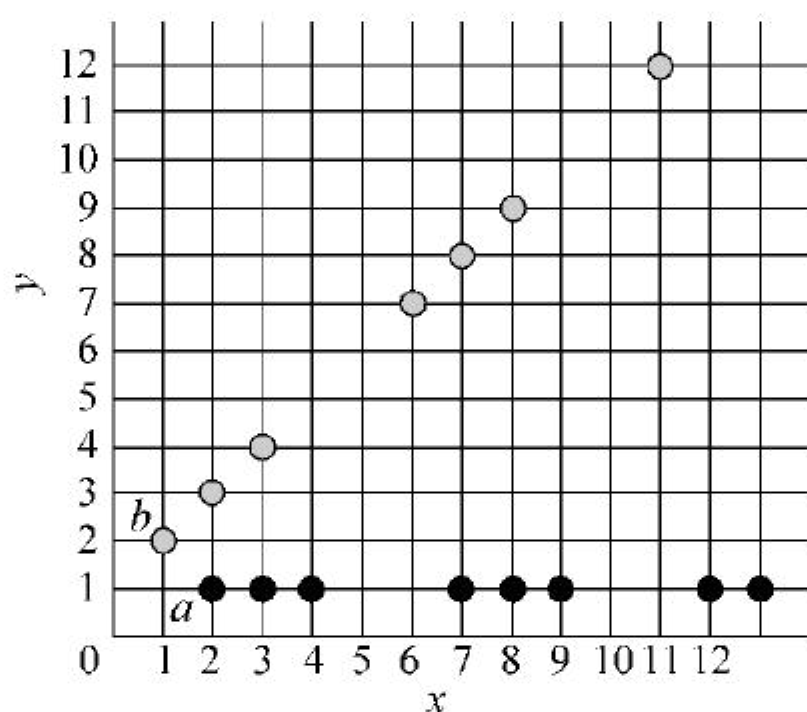


图 3.6-6 线型扫描转换显示

在扫描转换算法中，线型的显示可用像素模板(pixel mask)指定，像素模板是由数字 0 和 1 组成的位串，当对应为 1 时，显示像素，否则不显示。例如，模板 11100 可用来显示实线段长度为 3 个像素和中间空白段为 2 个像素的虚线，如图 3.6-6 所示。线型的像素模板用布尔值的数组来存放位串，在显示时，以数组的长度为周期进行重复。根据数组的长度以及数组中各位的数值，可以绘制不同类型的线型。

假设虚线的像素模板数组为：int lType[]={1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0}；数组长度为 16。

在画线时，从数组中寻找对应位置并判断是否写像素，参考代码如下：

```
if(lineType == 0)          //lineType 为线型标识符,0:直线,1:虚线
    pDC -> SetPixel(x,y,crColor);
else
    if(lType[it++ % 16] == 1)
        pDC -> SetPixel(x,y,crColor);
```

利用像素模板画线型存在的问题是，相同数量的像素数在不同方向上生成线段的长度是不同的，如图 3.6-6 所示，虚线在水平方向与在倾斜方向上二者的实线段和间隔的长度均不相同。

为了实现精确的线型定义，解决方法是按照直线的斜率来调整线型定义的像素模板数组中实线段和空白段的像素数目。或者对像素形成的线段的长度进行记录，对长度进行处理，而不是按像素的个数处理。

另一种精确定义线型的方法是，将线上的每一段实线段作为一段单独的线段，定位其首末点的坐标，再调用直线的扫描转换算法绘制；不过这样处理，具体实现时比较复杂。

3.7 反走样技术

利用前面介绍的各种扫描转换算法产生的直线、圆、椭圆以及多边形等基本图形，在边界部分或多或少都会呈现锯齿状，这是因为图形的数学描述是连续的，而最佳逼近图形的像

素点却是离散的，这种用离散量表示连续量引起的失真现象称为走样(aliasing)。走样是图形扫描转换的必然结果，走样现象不可避免，只能减轻。减少或克服走样效果的技术称为反走样技术，简称反走样(anti-aliasing)。

图形的走样有如下几种：

(1) 产生阶梯或锯齿形；

(2) 细节或纹理绘制失真；

(3) 狭小图形遗失；

(4) 实时动画忽隐忽现、闪烁跳跃。

反走样方法主要有两类。

第一类是超采样或称后置滤波。这类算法的基本思想着眼于提高分辨率，虽然采用高分辨率的光栅图形显示器也是一个选择，但它受到客观条件的限制，而且也不经济。因此，常采用软件方法实现，即：将低分辨率的图形像素划分为许多子像素，在较高分辨率上对各子像素的颜色值或灰度值进行计算，然后采用某种平均算法，将原像素内的各子像素的颜色值或灰度值的平均值作为该像素显示的颜色值或灰度值，在较低分辨率的光栅图形设备上进行显示。

可以用超采样方法来进行直线反走样。即：将每个像素分成 $n\times n$ 个子像素，然后在子像素级对直线进行光栅化，这样就可以得到每个像素中被激活的子像素的个数。如图 3.7-1 所示，粗实线正方形表示物理像素，虚线正方形表示子像素，阴影区域表示被激活的子像素。在 $n\times n$ 伪光栅上，可以光栅化的子像素最多为 n 个。每个物理像素的光强与其被激活的子像素数与 n 的比值成正比。假设一个物理像素中被激活的子像素有 m 个，其可能的最大光强为 $I_{\max}$，则该像素的显示亮度近似为 $\frac{m}{n^2}I_{\max}$；再取整，即可得到像素的显示灰度值。

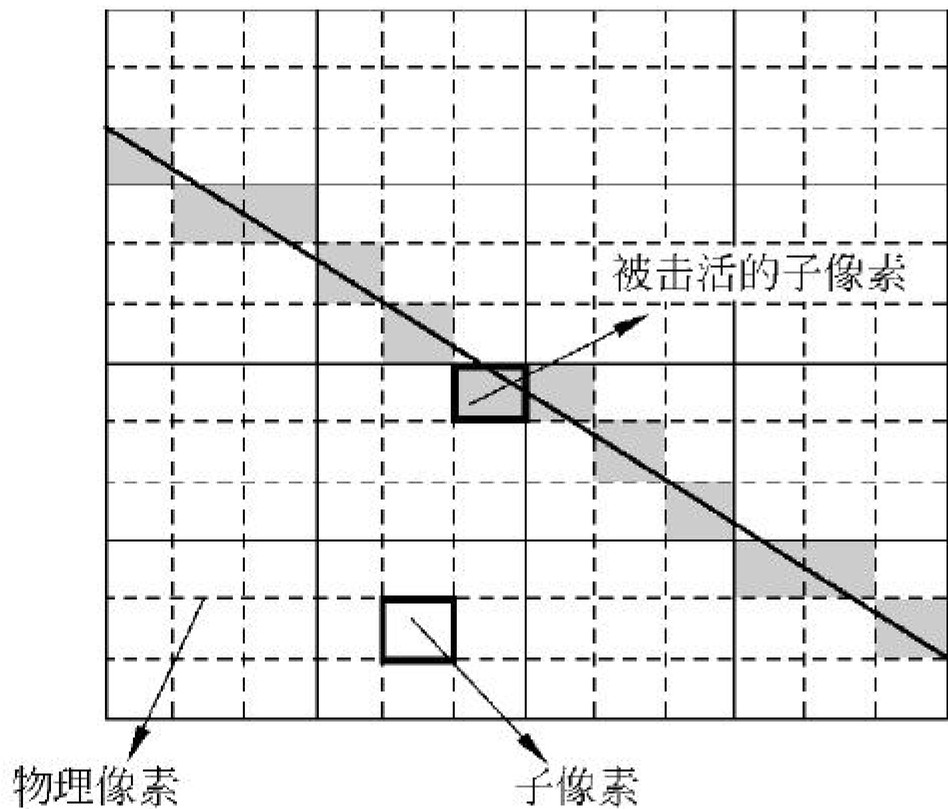


图 3.7-1　反走样细直线的超采样

第二类方法称为前置滤波。即：把像素作为一个有限区域而不是一个面积为零的点来处理。

假定每个像素都是一个面积等于 1 的小正方形区域，将直线段看作宽度为一个像素的狭长矩形，如图 3.7-2 所示，这时可以采用简单的区域采样方法进行反走样。当直线段的矩形边界与像素的边界有交时，求出两者相交区域的面积 A，然后根据相交区域面积的大小确

定该像素的亮度值。对于图 3.7-2 中的任何一个阴影像素而言，上述阴影面积 A 是介于 0～1 之间的正数，用它乘以像素的最大光强为 I_{max}，则该像素的光强

$$I = AI_{max}$$

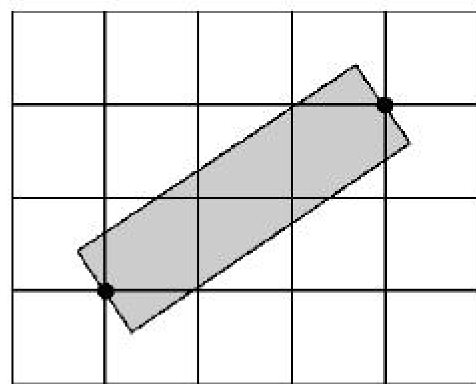


图 3.7-2　固定宽度直线

从采样理论的角度考虑，区域采样方法相当于使用盒式滤波器进行前置滤波后再采样。区域采样方法有两个缺点：①像素的亮度与相交区域的面积成正比，而与相交区域落在像素内的位置无关，这在某种程度上仍然会导致阶梯现象；②直线条上沿理想直线方向的相邻两个像素有时会有较大的亮度差，特别是当直线是一条接近水平或接近垂直的直线时，这种现象就会比较突出。

第 4 章

裁 剪

指定一个窗口作为边界,将窗口之外的图形裁掉,只保留窗口内的部分,这个过程称为裁剪(clipping)。裁剪是根据窗口参数确定窗口内那部分可见图形元素的一种处理过程。图形元素包括点、直线、多边形、字符等,裁剪是计算机图形学的重要理论基础之一。裁剪的边界可以是任意的多边形,最基本的是矩形裁剪窗口,例如显示器。直线段裁剪是图形裁剪的基础,裁剪的实质是判断直线段是否在裁剪窗口内,或者是否与窗口相交,如相交则进一步确定窗口内的部分。

4.1 点和直线的裁剪

4.1.1 点的裁剪

点的裁剪比较简单,假设裁剪窗口的 x 坐标区间为 (x_L, x_R),y 坐标区间为 (y_B, y_T)。则一点 (x, y) 可见的充要条件为

$$x_L \leqslant x \leqslant x_R$$
$$y_B \leqslant y \leqslant y_T$$

若其中任一条件不满足,则该点为不可见,应被裁剪掉。

4.1.2 直线裁剪

本书所提的直线指直线段,又称线段,直线裁剪是图形裁剪的基础。因为复杂的曲线可以通过折线段来近似,从而裁剪问题也可以化为直线段的裁剪问题。

线段裁剪的基本思路是:判断线段与裁剪窗口的位置关系,确定线段是完全可见、部分可见还是完全不可见;对部分可见线段,要求出它与窗口边框线的交点,输出落在窗口内线段的端点,并显示该线段。

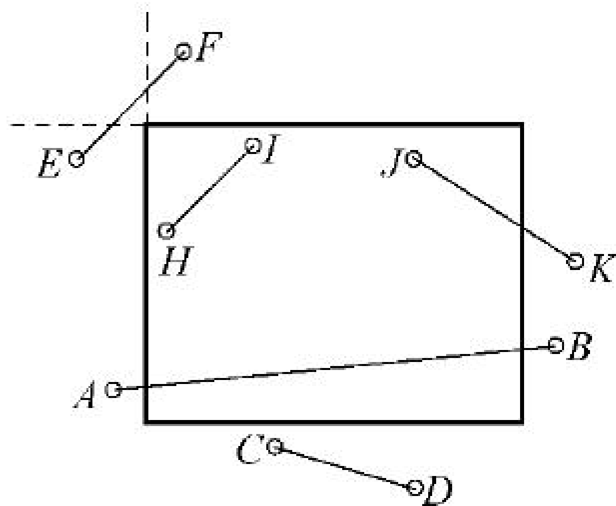


图 4.1-1　线段和裁剪窗口的位置关系

线段与裁剪窗口之间的位置关系如图 4.1-1 所示,有以下五种情况。

① 直线的两端点都在窗口内,完全可见,可被简单接受,

如图 4.1-1 中的 HI 线段。

② 直线的两端点都在窗口外，并且在窗口某边框线的同一侧，则完全不可见，可被简单裁掉，如图 4.1-1 中的 CD 线段。

③ 一端点在窗口内，另一端点在窗口外，如图 4.1-1 中的 JK 线段。

④ 两端点均在窗口外，但其中部横跨窗口，如图 4.1-1 中的 AB 线段。

⑤ 整条线在窗口外，且两端点不在窗口某边框线同侧，如图 4.1-1 中的 EF 线段。

其中对③④⑤三种情况均需要求出直线与窗口边框线的交点，并对交点的性质进行分析，对部分可见线段，裁掉外部一段，显示内部线段；对不可见线段，分析判断后全部裁掉。

直线裁剪算法有若干种，如：

① 逐边裁剪法：用窗口的每一段边界分别与直线段求交，裁剪；

② 矢量裁剪法：将屏幕分为 9 个区，分别处理；

③ 编码裁剪法(Cohen-Sutherland 算法)；

④ 中点分割算法：适合用硬件直接进行；

⑤ 直线方程法；

⑥ Cyrus-Beck 算法；

⑦ 梁友栋-Barskey 算法(Liang-Barsky 算法)。

Cohen-Sutherland 直线段裁剪算法是最早流行的裁剪算法，该算法中首先利用裁剪窗口的四个边界，将空间划分成 9 个区域，每个区域具有相同的四位编码，如图 4.1-2 所示。

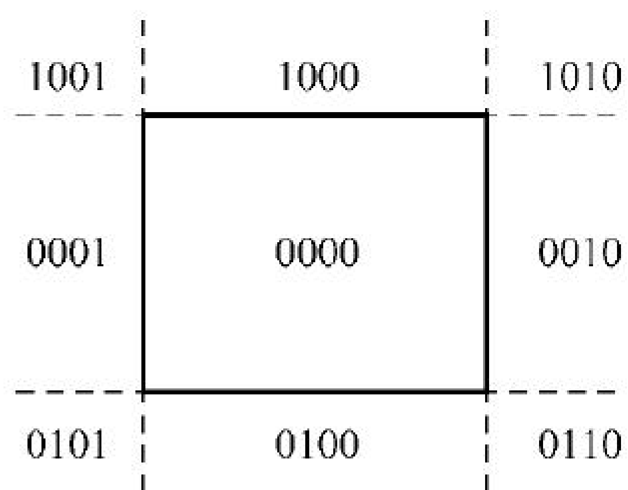


图 4.1-2 区域划分及编码

四位编码的顺序是上边界、下边界、右边界、左边界，裁剪窗口上边界线及其延长线上方区域的点，第一位都取 1，否则取 0；裁剪窗口下边界线及其延长线下方区域的点，第二位都取 1，否则取 0；裁剪窗口右边界线及其延长线右方区域的点，第三位都取 1，否则取 0；裁剪窗口左边界线及其延长线左方区域的点，第四位都取 1，否则取 0。因此各个区域的编码如图 4.1-2 所示。

线段的两个端点位于某一个区域时，就将该区域的代码赋予端点，即给端点编码。那么，可以对线段作如下判断和裁剪处理。

(1) 当线段两个端点的四位编码都是 0 时，表示线段位于窗口内，应保留。

(2) 当线段两个端点的四位编码不是 0 时，则线段两端点的四位编码按位进行逻辑与，如果不等于 0，表示两个端点的代码中有一个相同位同时为 1，则两个端点在裁剪窗口某个边界线外面的同一侧位置，那么，该线段位于窗口外，应舍弃。

(3) 当线段的两个端点不满足(1)和(2)的情况时，线段会与裁剪窗口的边界线或其延长线相交。需要逐边计算直线段与裁剪窗口四个边界直线的交点，如果存在交点，则交点将直线段分为两段，将其中位于裁剪窗口边界线外侧的一段舍弃，然后对交点进行编码，将另外一段作为新的直线段，继续对裁剪窗口剩余的其他边界线进行上述的判断处理，直至线段的两个端点都是 0 即在裁剪窗口内时，停止处理。

(4) 在逐边计算直线与裁剪窗口边界直线的交点时，可以利用直线段方程(针对一般位置直线，非特殊如水平和垂直位置直线)

$$\frac{y-y_1}{x-x_1}=\frac{y_2-y_1}{x_2-x_1}$$

其中，(x_1, y_1)，(x_2, y_2)是直线段的两个端点。当计算直线与左边界直线的交点时，将 $x=x_L$ 代入上述方程，即得交点坐标

$$\begin{cases} x = x_L \\ y = y_1 + \dfrac{(y_2-y_1)(x_L-x_1)}{x_2-x_1} \end{cases}$$

裁剪窗口其他边界直线和直线求交点的计算方法与上述类似。

例 4.1　对图 4.1-3 所示的线段 P_1P_2 进行窗口裁剪，步骤如下：

① 对 P_1 编码为 1000，对 P_2 编码为 0100；

② 对 P_1P_2 进行判断，不能被简单接受，也不能被简单裁剪掉；

③ 计算直线与边界线的交点，与上边界的交点为 P_3，并编码为 0000；

④ 对 P_3P_2 进行判断，不能被简单接受，也不能被简单裁剪掉；

⑤ 继续计算线段 P_3P_2 与其他界线的交点，得交点 P_4 并编码为 0000；

⑥ 对 P_3P_4 进行判断，能被简单接受，则保留线段 P_3P_4。

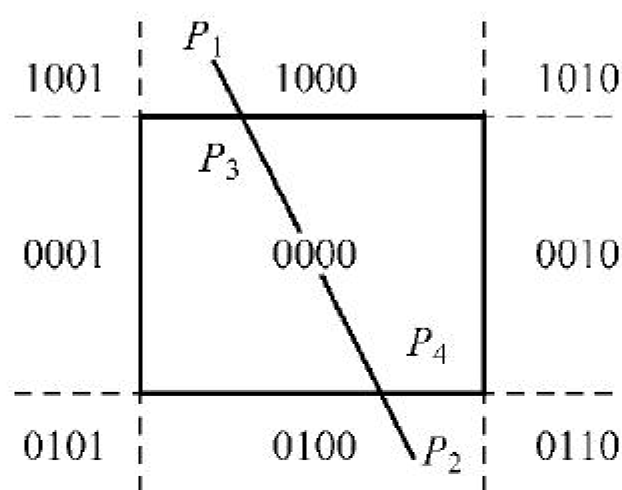


图 4.1-3　直线裁剪

Cohen-Sutherland 直线段裁剪算法的特点：用编码方法可快速判断线段的完全可见和显然不可见。其优点在于简单，易于实现。它可以简单地描述为将直线在窗口左边的部分删去，按左、右、下、上的顺序依次进行，处理之后，剩余部分就是可见的了。在这个算法中求交点是很重要的，它决定了算法的速度。另外，本算法对于其他形状的裁剪窗口未必同样有效。

在编程实现时，裁剪窗口可以定义为一个类对象结构：

```
class CCutRect:CDraw{
public:
    CCutRect(){
        flag = 0;
        xL = 0;
        xR = 0;
        yT = 0;
        yB = 0;}
    int xL,xR,yT,yB;
    int flag;                                    //是否有裁剪窗口 0: 没有,1: 有
};
```

针对多组线段的裁剪算法函数代码参考如下：

```
/**********************************************************************
OnRectCutlines: 线段裁剪算法
m_line_array: 被裁剪线段组及返回裁剪后的线段组; m_cutRect: 裁剪矩形窗口
**********************************************************************/
```

```
void OnRectCutlines(CArray<CLine,CLine> &m_line_array, CCutRect &m_cutRect,) {
    CLine line;
    CArray<CLine,CLine> m_line_array_tmp;              //存放裁剪后的临时线段组
    int code1[4],code2[4];
    while(m_line_array.GetSize()>0){                   //m_line_array 线段数组
        line = m_line_array.GetAt(0);                  //从线段组中取出一个线段
        m_line_array.RemoveAt(0);
        //1.线段端点编码
        Encode(line,m_cutRect,code1,code2);            //m_cutRect 为裁剪窗口
        //2.判断线段是否简单接受和放弃
if(code1[0]==0&&code1[1]==0&&code1[2]==0&&code1[3]==0&&code2[0]==0&&code2[1]==
0&&code2[2]==0&&code2[3]==0)   {
            m_line_array_tmp.Add(line);                //简单接受
            continue;
        }
        else if((code1[0]&code2[0])||(code1[1]&code2[1])||(code1[2]&code2[2])||(code1[3]
&code2[3])){
            continue;                                  //在同一侧,则放弃
        }
        //非简单接受和简单放弃,则求交点并获得裁剪窗口内的线段
        InterPt(m_line_array_tmp,line,m_cutRect,code1,code2);  }
    m_line_array.Append(m_line_array_tmp);             //裁剪后的线段组赋给原线段组
    m_line_array_tmp.RemoveAll();
    Invalidate();                                      //显示
}
```

其中,线段端点的编码函数为:

```
/* line: 线段; m_cutRect: 裁剪矩形窗口; code1: 端点 1 的编码; code2: 端点 2 的编码 */
void Encode(CLine &line,CCutRect m_cutRect,int * code1,int * code2){
    if(line.pt1.y<m_cutRect.yT) code1[0]=1;else code1[0]=0;
    if(line.pt1.y>m_cutRect.yB) code1[1]=1;else code1[1]=0;
    if(line.pt1.x>m_cutRect.xR) code1[2]=1;else code1[2]=0;
    if(line.pt1.x<m_cutRect.xL) code1[3]=1;else code1[3]=0;

    if(line.pt2.y<m_cutRect.yT) code2[0]=1;else code2[0]=0;
    if(line.pt2.y>m_cutRect.yB) code2[1]=1;else code2[1]=0;
    if(line.pt2.x>m_cutRect.xR) code2[2]=1;else code2[2]=0;
    if(line.pt2.x<m_cutRect.xL) code2[3]=1;else code2[3]=0;
}
```

对于非简单接受和简单放弃的情况,求交点并获得裁剪窗口内线段的函数如下:

```
/* m_line_array: 线段组; line: 线段; m_cutRect: 裁剪矩形窗口; code1: 端点 1 的编码; code2:
端点 2 的编码 */
void InterPt(CArray<CLine,CLine> &m_line_array,CLine &line,CCutRect m_cutRect,int * code1,
int * code2){
    CPoint pt;
    if(code1[0]||code2[0])  {                          //上边界
        if(code1[0]&&code2[0]) return;                 //在裁剪窗口外,则删除
        //与 yt 求交点
        pt.y = m_cutRect.yT;
```

```
pt.x = line.pt1.x + (double)(line.pt2.x - line.pt1.x)/(double)(line.pt2.y - line.pt1.y) * (pt.
y - line.pt1.y);
        if(code1[0] == 1){
            line.pt1 = pt;
            code1[0] = 0;
            if(pt.x < m_cutRect.xL) code1[3] = 1;
            else if(pt.x > m_cutRect.xR) code1[2] = 1;
        }
        else  {
            line.pt2 = pt;
            code2[0] = 0;
            if(pt.x < m_cutRect.xL) code2[3] = 1;
            else if(pt.x > m_cutRect.xR) code2[2] = 1;
        }
    }
    if(code1[1]||code2[1])  {                               //下边界
        if(code1[1]&&code2[1]) return;                      //在裁剪窗口外,则删除
        //与 yt 求交点
        pt.y = m_cutRect.yB;
pt.x = line.pt1.x + (double)(line.pt2.x - line.pt1.x)/(double)(line.pt2.y - line.pt1.y) * (pt.
y - line.pt1.y);
        if(code1[1] == 1){
            line.pt1 = pt;
            code1[1] = 0;
            if(pt.x < m_cutRect.xL) code1[3] = 1;
            else if(pt.x > m_cutRect.xR) code1[2] = 1;
        }
        else  {
            line.pt2 = pt;
            code2[1] = 0;
            if(pt.x < m_cutRect.xL) code2[3] = 1;
            else if(pt.x > m_cutRect.xR) code2[2] = 1;
        }
    }
    if(code1[2]||code2[2])  {                               //右边界
        if(code1[2]&&code2[2]) return;                      //在裁剪窗口外,则删除
        pt.x = m_cutRect.xR;                                //与 xL 求交点
pt.y = (int)line.pt1.y + (m_cutRect.xR - line.pt1.x) * (double)(line.pt2.y - line.pt1.y)/
(double)(line.pt2.x - line.pt1.x);
        if(code1[2] == 1){
            line.pt1 = pt;
            code1[2] = 0;
            if(pt.y < m_cutRect.yT) code1[0] = 1;
            else if(pt.y > m_cutRect.yB) code1[1] = 1;
        }
        else  {
            line.pt2 = pt;
            code2[2] = 0;
            if(pt.y < m_cutRect.yT)code2[0] = 1;
            else if(pt.y > m_cutRect.yB)code2[1] = 1;
        }
```

```
    }
    if(code1[3]||code2[3])  {                                   //左边界
        if(code1[3]&&code2[3]) return;                          //在裁剪窗口外,则删除
        //与 xL 求交点
        pt.x = m_cutRect.xL;
pt.y = (int)line.pt1.y + (m_cutRect.xL - line.pt1.x) * (line.pt2.y - line.pt1.y)/(double)
(line.pt2.x - line.pt1.x);
        if(code1[3] == 1){
            line.pt1 = pt;
            code1[3] = 0;
            if(pt.y < m_cutRect.yT) code1[0] = 1;
            else if(pt.y > m_cutRect.yB) code1[1] = 1;
        }
        else  {
            line.pt2 = pt;
            code2[3] = 0;
            if(pt.y < m_cutRect.yT)code2[0] = 1;
            else if(pt.y > m_cutRect.yB)code2[1] = 1;
        }
    }
    //再判断通过求交后的线段的位置
    if(code1[0] == 0&&code1[1] == 0&&code1[2] == 0&&code1[3] == 0&&code2[0] == 0&&code2[1] =
= 0&&code2[2] == 0&&code2[3] == 0)  {
        m_line_array.Add(line);                                 //简单接受
    }
    else if((code1[0]&code2[0])||(code1[1]&code2[1])||(code1[2]&code2[2])||(code1[3]
&code2[3])){
        return;                                                 //在同一侧,则放弃
    }
    else  {
        InterPt(m_line_array,line,m_cutRect,code1,code2); //再递归调用本函数
    }
}
```

图 4.1-4 所示为上述代码实现的线段的裁剪效果图。本算法的特点为用编码方法可快速判断线段的完全可见和显然不可见。其优点在于简单,易于实现。它可以简单地描述为将直线在窗口左边的部分删去,按左、右、下、上的顺序依次进行,处理之后,剩余部分就是可见的了。在这个算法中求交点是很重要的,它决定了算法的速度。另外,本算法对于其他形状的窗口未必同样有效。

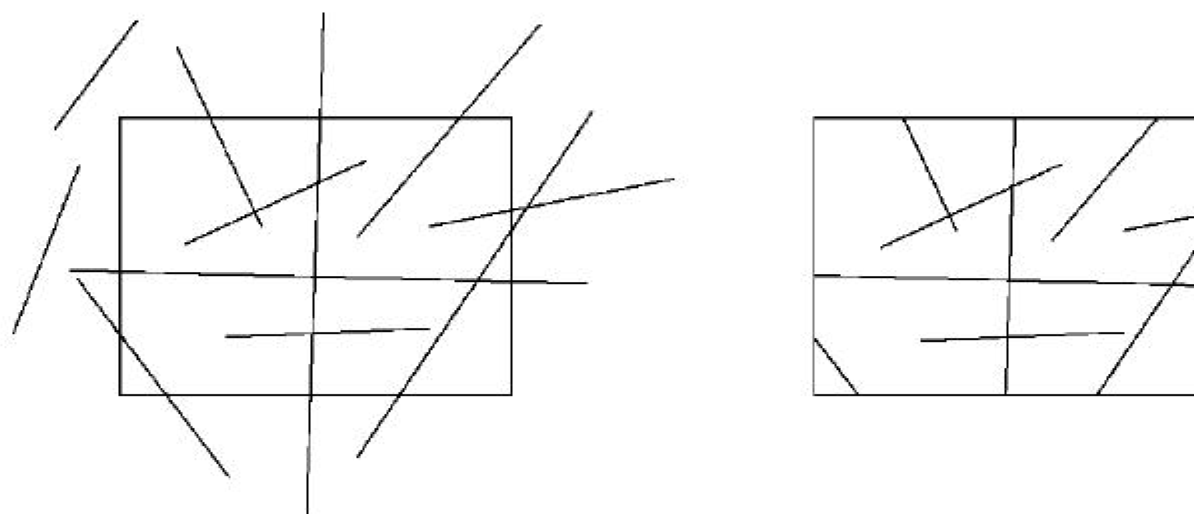


图 4.1-4 Cohen-Sutherland 直线裁剪算法

4.2 多边形裁剪

4.2.1 多边形裁剪概述

多边形是由首尾相接的直线段组成的封闭图形，所以，多边形裁剪看似和多条直线段相对裁剪窗口的裁剪方法类似，但是，这样裁剪后，多边形的边界将可能不再是封闭的状态，多边形裁剪的结果将不再是多边形，我们希望多边形裁剪后仍然还是个封闭的形状。当多边形边界裁剪后不再封闭时，需要用窗口边界的恰当部分来封闭它，所以，多边形裁剪要解决的第一个问题是：通过裁剪，不仅要保持窗口内多边形的边界部分，而且要将裁剪窗口的有关部分按一定次序插入多边形的保留边界之间，从而使裁剪后的多边形之边仍旧保持封闭状态，使原来的多边形填充算法得以正确实现。多边形裁剪要解决的第二个问题是：裁剪后如果出现了多个多边形区域，多个区域之间是否有关联？如何确定边界？不同的裁剪算法可得到不同的处理结果。

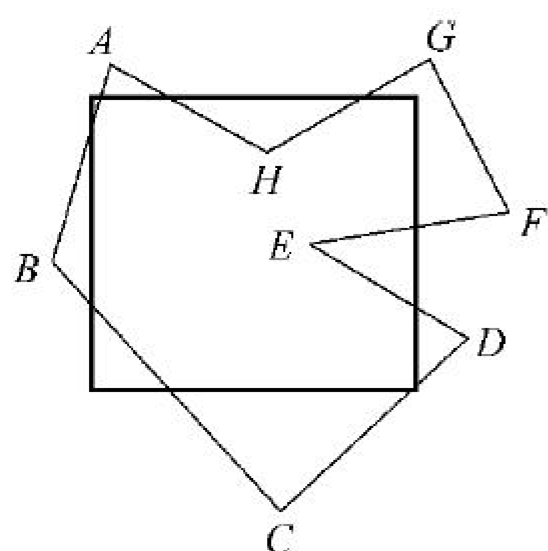
由于多边形有凸多边形、凹多边形以及含内环的多边形等多种类型，裁剪窗口也可以是各种类型的多边形，因此，多边形裁剪相比直线的裁剪要复杂得多，需要考虑各种情况。某种多边形裁剪算法可能只对某种类型的多边形进行裁剪，裁剪结果是正确的，如果换做另外一种类型的多边形，该裁剪算法可能并没有效果。适合矩形裁剪窗口的算法有 Sutherland-Hodgman 算法（逐边裁剪法）、Liang-Barsky 算法、中点分割算法；适合凸多边形裁剪窗口的算法有 Cyrus-Beck 线裁剪算法、Sutherland-Hodgman 逐边裁剪算法、外接矩形判别法以及分区判断直接裁剪法；适合任意多边形裁剪窗口的有 Weiler-Atherton 算法（双边裁剪法），等等。其中，Sutherland-Hodgman 逐边裁剪算法适用于被裁剪多边形可以是任意凸多边形或凹多边形、裁剪窗口不局限于矩形也可以是任意凸多边形的情况，可以很好地诠释多边形裁剪的原理和方法。而 Weiler-Atherton 双边裁剪算法则适用于裁剪窗口是任意形状的多边形情况。本书限于篇幅，仅讨论上述两种裁剪方法，其他裁剪方法不再赘述，关于多边形新的裁剪算法，目前仍然在研究中。

4.2.2 矩形及凸多边形裁剪窗口裁剪

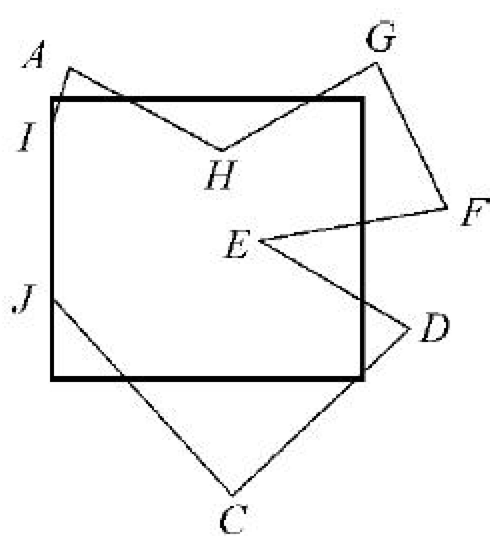
当裁剪窗口是矩形或其他凸多边形时，可以采用 Sutherland-Hodgman 算法（简称 S-H 算法）来实现。Sutherland-Hodgman 算法也叫逐边裁剪法，该算法是 Sutherland 和 Hodgman 在 1974 年提出的，这种算法采用了分割处理、逐边裁剪的方法。S-H 逐边裁剪法的基本思路是：把整个多边形先相对于裁剪窗口的第一条边界线进行裁剪，形成一个新的多边形，把它作为中间多边形，然后再把这个中间多边形用裁剪窗口的第二条边界线进行裁剪，又形成一个新的中间多边形，接着用窗口的第三、第四条边界线依次进行裁剪。以此类推，最后形成整个多边形经过裁剪窗口边界线裁剪的最终结果。

对于如图 4.2-1(a)所示的多边形 $ABCDEFGH$ 及矩形裁剪窗口，首先，沿左裁剪边界裁剪，获得一个中间多边形 $AIJCDEFGH$，如图 4.2-1(b)所示。然后对该中间多边形沿上

裁剪边界裁剪，获得新中间多边形 $KIJCDEFNMHL$，如图 4.2-2 所示。再对该新中间多边形沿右裁剪边界裁剪，获得新中间多边形 $KIJCPOENRMHL$，如图 4.2-3 所示。再对该新中间多边形沿右裁剪边界裁剪，获得新中间多边形 $KIJQRPOENRMHL$，如图 4.2-4 所示。



(a) 多边形及裁剪窗口



(b) 第一个边界裁剪

图 4.2-1　矩形裁剪窗口裁剪

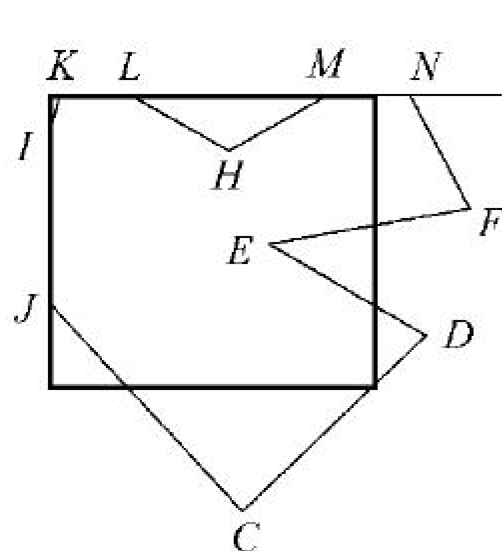


图 4.2-2　第二个边界裁剪

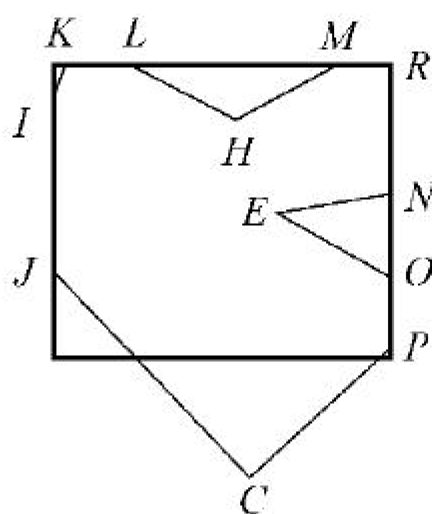


图 4.2-3　第三个边界裁剪

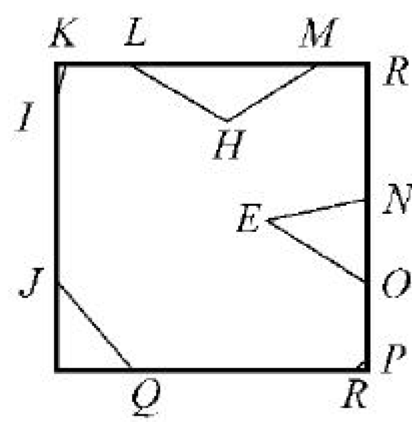
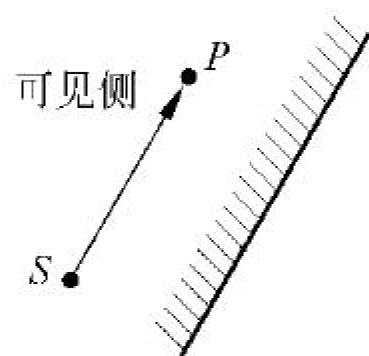


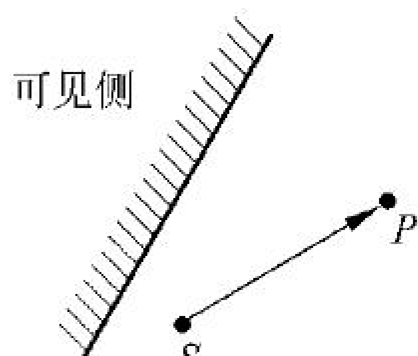
图 4.2-4　第四个边界裁剪

在具体实施该算法时，应把多边形作为顶点序列而不是点阵序列来处理，即算法的输入是以顶点序列表示的多边形。算法的输出也是一个顶点序列，构成一个或多个新的多边形，裁剪算法的任务就是求得这些新的多边形顶点序列。

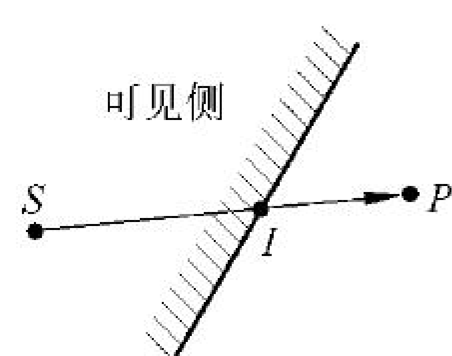
在依次对多边形的每条边与窗口裁剪边界进行处理时，首先需要分析这条边与裁剪边界的位置关系。假设对多边形的某一条边，分别以 S 和 P 表示它的起点和终点，那么这条边和裁剪线的位置关系会有如图 4.2-5 所示的四种可能情况。



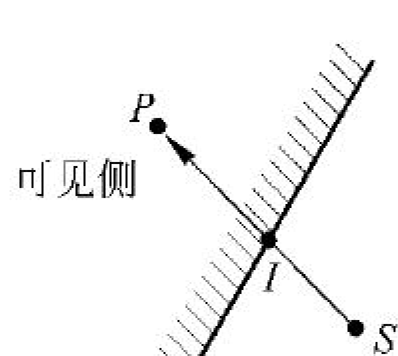
(a) 完全可见



(b) 完全不可见



(c) 部分可见(离开可见侧)



(d) 部分可见(进入可见侧)

图 4.2-5　多边形一条边与裁剪边界的位置关系

多边形与每一条窗边相交，生成新的多边形顶点序列的过程，是一个对多边形各顶点依次处理的过程。设当前处理的顶点为 P，先前顶点为 S，则多边形各顶点的处理规则如下。

如果 S、P 均在裁剪窗口边界之内侧，那么，将 P 保存。

如果 S 在裁剪窗口边界内侧，P 在外侧，那么，求出 SP 边与裁剪窗口边界的交点 I，保存 I，舍去 P。

如果 S、P 均在裁剪窗口边界外侧，那么，舍去 P。

如果 S 在裁剪窗口边界外侧，P 在内侧，那么，求出 SP 边与裁剪窗口边界的交点 I，依次保存 I 和 P。

基于这四种情况，可以归纳对当前点 P 的处理方法为：①P 在裁剪窗口边界内侧，则保存 P；否则不保存。②P 和 S 在裁剪窗口边界非同侧，则求交点 I，并将 I 保存，并插入 P 之前，或 S 之后。

对于裁剪窗口是矩形的情况，上述的多边形逐边裁剪算法函数代码参考如下：

```
/*******************************************************************
PolylineCut：多边形逐边裁剪算法
m_PolyLine_array：多边形边数组；m_cutRect：矩形裁剪窗口
*******************************************************************/
    void PolylineCut(CArray<CLine,CLine> &m_PolyLine_array,CCutRect m_cutRect){
        CArray<CPoint,CPoint> m_point_Array,m_point_Array1,m_point_Array2,m_point_
Array3,m_point_Array4;                              //构造中间多边形
    //把边变成顶点序列
    CLine line;
    for(int j=0;j<m_PolyLine_array.GetSize();j++)  {
        line=m_PolyLine_array.GetAt(j);
        m_point_Array.Add(line.pt1);
    }
    m_point_Array.Add(m_point_Array.GetAt(0));  //首先把首点加到最后一点,使首尾相接,最
                                                //后删除
    int x_ledge,y_ledge;
    x_ledge=m_cutRect.xL;
    int a, b,c;
    double m_dblX,m_dblY;
    int x0,x1,y0,y1;
    CPoint newPoint;
    //1.xmin裁剪
    for(int i=0;i<m_point_Array.GetSize()-1;i++)  {
if(m_point_Array.GetAt(i).x<x_ledge&&m_point_Array.GetAt(i+1).x<x_ledge)
            continue;                               //继续,放弃该边
        else if(m_point_Array.GetAt(i).x>=x_ledge&&m_point_Array.GetAt(i+1).x>=x_ledge){
            //接受
            m_point_Array1.Add(m_point_Array.GetAt(i));
        }
        else  {
            x0=m_point_Array.GetAt(i).x;                //一边一个,则计算交点并插入
            x1=m_point_Array.GetAt(i+1).x;
            y0=m_point_Array.GetAt(i).y;
            y1=m_point_Array.GetAt(i+1).y;
            a=y0-y1;
            b=x1-x0;
            c=x0*y1-x1*y0;
            m_dblY=(-1)*(double(a*x_ledge+c))/(double)b;
```

```
                y_ledge = (int)(m_dblY + 0.5);
                newPoint.x = x_ledge;
                newPoint.y = y_ledge;
                //计算插入顺序
                if(m_point_Array.GetAt(i).x < x_ledge)
                    m_point_Array1.Add(newPoint);            //插入交点
                else  {
                    //先插入端点再插入交点
                    m_point_Array1.Add(m_point_Array.GetAt(i));
                    m_point_Array1.Add(newPoint);
                }
            }
        }
        //2. ymin裁剪
        y_ledge = m_cutRect.yT;
        m_point_Array1.Add(m_point_Array1.GetAt(0));
        for(i = 0;i < m_point_Array1.GetSize() - 1;i++){
    if(m_point_Array1.GetAt(i).y < y_ledge&&m_point_Array1.GetAt(i + 1).y < y_ledge)
                continue;                                    //继续,放弃该边
            else if(m_point_Array1.GetAt(i).y >= y_ledge&&m_point_Array1.GetAt(i + 1).y >= y_ledge)
                m_point_Array2.Add(m_point_Array1.GetAt(i)); //接受
            else  {
                //一边一个,则计算交点并插入
                x0 = m_point_Array1.GetAt(i).x;
                x1 = m_point_Array1.GetAt(i + 1).x;
                y0 = m_point_Array1.GetAt(i).y;
                y1 = m_point_Array1.GetAt(i + 1).y;
                a = y0 - y1;
                b = x1 - x0;
                c = x0 * y1 - x1 * y0;
                m_dblX = ( - 1) * (double(b * y_ledge + c))/(double)a;
                x_ledge = (int)(m_dblX + 0.5);
                newPoint.x = x_ledge;
                newPoint.y = y_ledge;
                //计算插入顺序
                if(m_point_Array1.GetAt(i).y < y_ledge)      //先插入交点
                    m_point_Array2.Add(newPoint);
                else  {
                    //先插入端点,再插入交点
                    m_point_Array2.Add(m_point_Array1.GetAt(i));
                    m_point_Array2.Add(newPoint);
                }
            }
        }
        //3. xmax裁剪
        x_ledge = m_cutRect.xR;
        m_point_Array2.Add(m_point_Array2.GetAt(0));
        for(i = 0;i < m_point_Array2.GetSize() - 1;i++){
    if(m_point_Array2.GetAt(i).x > x_ledge&&m_point_Array2.GetAt(i + 1).x > x_ledge)
                continue;                                    //继续,放弃该边
            else if(m_point_Array2.GetAt(i).x <= x_ledge&&m_point_Array2.GetAt(i + 1).x <= x_ledge)
```

```
            m_point_Array3.Add(m_point_Array2.GetAt(i)); //接受
        else {//一边一个,则计算交点并插入
            x0 = m_point_Array2.GetAt(i).x;
            x1 = m_point_Array2.GetAt(i+1).x;
            y0 = m_point_Array2.GetAt(i).y;
            y1 = m_point_Array2.GetAt(i+1).y;
            a = y0 - y1;
            b = x1 - x0;
            c = x0 * y1 - x1 * y0;
            m_dblY = (-1) * (double(a * x_ledge + c))/(double)b;
            y_ledge = (int)(m_dblY + 0.5);
            newPoint.x = x_ledge;
            newPoint.y = y_ledge;
            //计算顺序
            if(m_point_Array2.GetAt(i).x > x_ledge)
                m_point_Array3.Add(newPoint);           //插入交点
            else {//先插入端点
                m_point_Array3.Add(m_point_Array2.GetAt(i));
                m_point_Array3.Add(newPoint);
            }
        }
    }
    //4. ymax裁剪
    y_ledge = m_cutRect.yB;
    m_point_Array3.Add(m_point_Array3.GetAt(0));
    for(i = 0;i < m_point_Array3.GetSize() - 1;i++){
if(m_point_Array3.GetAt(i).y > y_ledge&&m_point_Array3.GetAt(i+1).y > y_ledge)
            continue;                                   //继续,放弃该边
        else if(m_point_Array3.GetAt(i).y <= y_ledge&&m_point_Array3.GetAt(i+1).y <= y_ledge)
            m_point_Array4.Add(m_point_Array3.GetAt(i)); //接受
        else {//一边一个,则计算交点并插入
            x0 = m_point_Array3.GetAt(i).x;
            x1 = m_point_Array3.GetAt(i+1).x;
            y0 = m_point_Array3.GetAt(i).y;
            y1 = m_point_Array3.GetAt(i+1).y;
            a = y0 - y1;
            b = x1 - x0;
            c = x0 * y1 - x1 * y0;
            m_dblX = (-1) * (double(b * y_ledge + c))/(double)a;
            x_ledge = (int)(m_dblX + 0.5);
            newPoint.x = x_ledge;
            newPoint.y = y_ledge;
            //计算顺序
            if(m_point_Array3.GetAt(i).y > y_ledge)
                m_point_Array4.Add(newPoint);           //插入交点
            else {//先插入端点
                m_point_Array4.Add(m_point_Array3.GetAt(i));
                m_point_Array4.Add(newPoint);
            }
        }
    }
```

```
    //重新加入多边形边中
    m_PolyLine_array.RemoveAll();
    for(j = 0;j < m_point_Array4.GetSize() - 1;j++){
        line.pt1 = m_point_Array4.GetAt(j);
        line.pt2 = m_point_Array4.GetAt(j + 1);
        m_PolyLine_array.Add(line);
    }
    //首尾相接
    line.pt1 = m_point_Array4.GetAt(m_point_Array4.GetSize() - 1);
    line.pt2 = m_point_Array4.GetAt(0);
    m_PolyLine_array.Add(line);
}
```

如图 4.2-6 所示为对于一多边形和矩形裁剪窗口利用上述逐边裁剪法裁剪的结果，以及对裁剪后的多边形进行扫描转换的效果图。

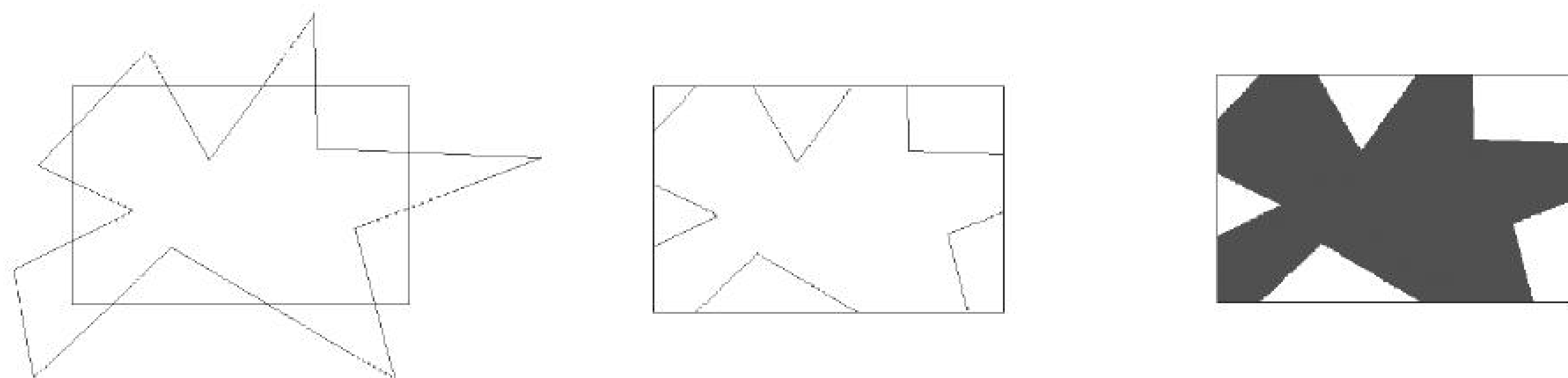


图 4.2-6 矩形裁剪窗口裁剪多边形

当裁剪窗口不是矩形，而是一般任意多边形时，那么利用上述的逐边裁剪法分析判断被裁剪多边形的顶点在裁剪窗口边界的哪一侧以及计算交点时，不能像矩形窗口那样通过简单判断和计算来实现。

为了判断多边形顶点与裁剪边界的位置关系，首先需要将裁剪多边形的每条边设置为有向线段，裁剪多边形的外环按顺时针方向首尾相接，裁剪多边形的内环按逆时针方向首尾相接。当裁剪多边形是凸多边形时，在判断多边形走向时，可以利用数学上向量的叉积来判断(可称之为有向线段叉积判别法)。

假设多边形有向顶点序列为 $P_0P_1P_2P_3P_4$，每个顶点的坐标为 (x_i, y_i)，$i=0,1,\cdots,4$，如图 4.2-7 所示，则相邻有向线段$\overrightarrow{P_0P_1}\times\overrightarrow{P_1P_2}$叉积的方向垂直于多边形各顶点所在的平面 xOy，即在坐标轴 z 方向上，并遵守叉积的右手定则。当顶点序列方向是顺时针时，$\overrightarrow{P_0P_1}\times\overrightarrow{P_1P_2}$叉积的方向和 z 轴的正方向相反；当顶点序列是逆时针方向时，$\overrightarrow{P_0P_1}\times\overrightarrow{P_1P_2}$叉积方向和 z 轴正方向相同。$\overrightarrow{P_0P_1}\times\overrightarrow{P_1P_2}$的叉积为

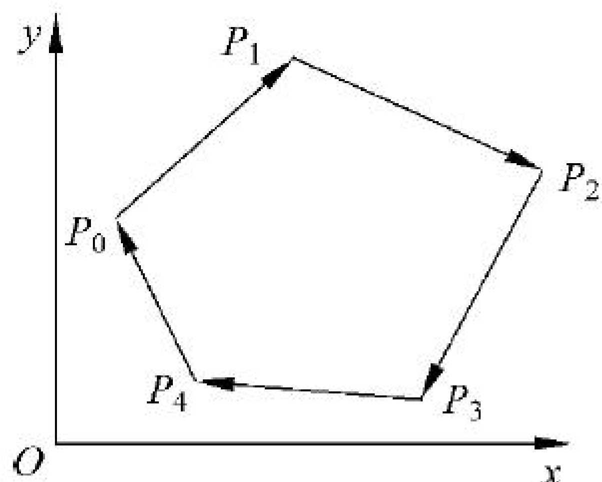


图 4.2-7 有向多边形

$$\begin{bmatrix} i & j & k \\ x_1-x_0 & y_1-y_0 & 0 \\ x_2-x_1 & y_2-y_1 & 0 \end{bmatrix}$$

在 z 轴的分量为 $(x_1-x_0)(y_2-y_1)-(x_2-x_1)(y_1-y_0)$。由此，可以通过计算叉积在 z 轴的分量来判断和设置多边形的顶点序列的方向。

在判断多边形某顶点在裁剪窗口边界外侧或者内侧时，也

可以利用叉积的正负来进行。如图4.2-8所示，AB为裁剪多边形窗口的一个有向边，P为被裁剪多边形的一个顶点。

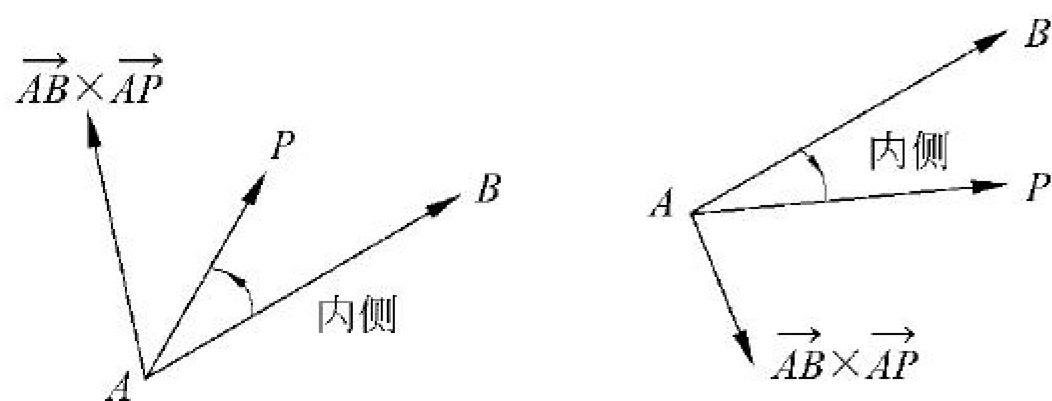


图4.2-8　顶点与裁剪边界

当顶点P在裁剪多边形外侧时，根据叉积的右手定则，$\overrightarrow{AB}\times\overrightarrow{AP}$的叉积的方向与$z$轴正方向相同；当顶点$P$在裁剪多边形内侧时，$\overrightarrow{AB}\times\overrightarrow{AP}$的叉积的方向与$z$轴正方向相反。因此，在逐边裁剪时，即可利用叉积的正负号来判断顶点在裁剪边界的哪一侧，从而进一步判断顶点对应的多边形的边在裁剪边界的哪一侧以及与裁剪边界相交的情况，该顶点是保存还是舍弃。

当多边形的边和裁剪边界相交时，首先需要计算交点。可利用直线的隐式方程求直线的交点，设多边形某边的两个端点为$P_s(x_s,y_s)$，$P_e(x_e,y_e)$，裁剪多边形的裁剪边的两个端点为$P_a(x_a,y_a)$，$P_b(x_b,y_b)$，联立求解以下两个直线方程：

$$\begin{cases}a_1x+b_1y+c_1=0\\a_2x+b_2y+c_2=0\end{cases}$$

式中，$a_1=y_s-y_e$，$b_1=x_e-x_s$，$c_1=x_sy_e-x_ey_s$，$a_2=y_a-y_b$，$b_2=x_b-x_a$，$c_2=x_ay_b-x_by_a$。

解得

$$\begin{cases}x=\dfrac{c_2b_1-c_1b_2}{a_1b_2-a_2b_1}\\[2ex]y=\dfrac{c_1a_2-c_2a_1}{a_1b_2-a_2b_1}\end{cases}$$

上述算法的凸多边形窗口的多边形裁剪函数代码参考如下：

```
/*********************************************************************
Polyline_Cut: 凸多边形裁剪窗口裁剪算法
m_PolyLine_array:首尾相接的任意多边形边数组; ringFlag = 0: 外环,1: 内环; m_cutPolyLine:
凸多边形裁剪窗口
*********************************************************************/
    void Polyline_Cut(CArray<CLine,CLine>&m_PolyLine_array,int ringFlag,CPolyLine &m_
cutPolyLine){
        CArray<CPoint,CPoint>m_point_Array,m_cut_point_Array,m_point_Array1;
    //1.多边形改为顶点序列
    CLine line1;
    for(int jk = 0;jk<m_PolyLine_array.GetSize();jk++){
        line1 = m_PolyLine_array.GetAt(jk);
        m_point_Array.Add(line1.pt1);}
    m_point_Array.Add(m_point_Array.GetAt(0));            //首尾相接
    //2.判断裁剪多边形是否为顺时针,如不是,则把顶点序列改为顺时针
SortForPolyline(m_cutPolyLine.m_PolyLine_array_Out,ringFlag ,m_cut_point_Array);
```

```
//3.循环从裁剪多边形中逐次取出一条边,作为裁剪边界
CPoint ptA,ptB,interPt;                              //,pt0,pt1;
int inOroutflag = 0;                                 //0: 外侧,1: 内侧
for(int j = 0;j < m_cut_point_Array.GetSize() - 1;j++){
    ptA = m_cut_point_Array.GetAt(j);
    ptB = m_cut_point_Array.GetAt(j + 1);
    //取多边形第一个顶点判断在内侧还是外侧
    pt0 = m_point_Array.GetAt(0);
    if(VectorXVector(ptB,ptA,pt0) * ( - 1)> 0)       //外侧
        inOroutflag = 0;
    else  {
        inOroutflag = 1;                             //内侧
        m_point_Array1.Add(pt0);                     //加入新的顶点序列
    }
    //循环从多边形中取出一个顶点,判断如何取舍
    for(int k = 1;k < m_point_Array.GetSize();k++){
        pt1 = m_point_Array.GetAt(k);
        //取顶点判断在内侧还是外侧
        if(VectorXVector(ptB,ptA,pt1) * ( - 1)> 0){  //外侧
            //如果上一个顶点也在外侧,则舍去
            if(inOroutflag == 0) {
                pt0 = pt1;
                continue;
            }
            else {      //上一个顶点在内侧,则有交点,计算交点,并将交点插入顶点序列,
                        //舍去当前顶点,并设新的 inOroutflag = 0
                        //计算交点
                InterPtToPt(ptA,ptB,pt0,pt1,interPt);//插入交点
                m_point_Array1.Add(interPt);         //加入新的顶点序列
                inOroutflag = 0;
                pt0 = pt1;
            }
        }
        else{//内侧
            //如果上一个交点也在内侧,则加入新顶点序列
            if(inOroutflag == 1){                    //内侧
                m_point_Array1.Add(pt1);             //加入新的顶点序列
                pt0 = pt1;
                continue;
            }
            else{//上一个顶点在外侧,有交点,将交点插入顶点序列再插入顶点
                InterPtToPt(ptA,ptB,pt0,pt1,interPt);//计算交点
                m_point_Array1.Add(interPt);         //加入新的顶点序列
                inOroutflag = 1;                     //设置新的 inOroutflag = 1 在内侧
                m_point_Array1.Add(pt1);             //顶点加入新的顶点序列
                pt0 = pt1;
            }
        }
    }
    m_point_Array.RemoveAll();
    m_point_Array.Append(m_point_Array1);
```

```
            m_point_Array1.RemoveAll();
        }
        //重新加入多边形边中
        m_PolyLine_array.RemoveAll();
        for(j = 0;j < m_point_Array.GetSize() - 1;j++){
            line.pt1 = m_point_Array.GetAt(j);
            line.pt2 = m_point_Array.GetAt(j + 1);
            m_PolyLine_array.Add(line);
        }
        //首尾相接
        line.pt1 = m_point_Array.GetAt(m_point_Array.GetSize() - 1);
        line.pt2 = m_point_Array.GetAt(0);
        m_PolyLine_array.Add(line);
}
```

其中,多边形顶点的排序函数如下:

```
/ * m_PolyLine_array: 多边形; ringFlag: 多边形方向; m_point_Array: 排序后的顶点序列 * /
void SortForPolyline(CArray < CLine, CLine > &m_PolyLine_array, int ringFlag, CArray < CPoint,
CPoint > &m_point_Array){
    //1.判断多边形外环是否顺时针方向,如不是,把顶点序列改为正确顺序,外环顺时针,内环逆时针
    CPoint pt0,pt1,pt2;
    pt0 = m_PolyLine_array.GetAt(0).pt1;
    pt1 = m_PolyLine_array.GetAt(0).pt2;
    pt2 = m_PolyLine_array.GetAt(1).pt2;
    CLine line;
    CPoint intPt;
    if(VectorXVector(pt0,pt1,pt2)< 0){                  //多边形初始是顺时针
        if(ringFlag == 0){                              //是外环,顶点序列按初始边的顺序排序
            for(int j = 0;j < m_PolyLine_array.GetSize();j++){
                line = m_PolyLine_array.GetAt(j);
                intPt.x = line.pt1.x;
                intPt.y = line.pt1.y;
                m_point_Array.Add(intPt);
            }
        }
        else {
            //是内环,顶点序列按初始边的反方向顺序排序
            for(int j = m_PolyLine_array.GetSize() - 1;j >= 0;j -- )  {
                line = m_PolyLine_array.GetAt(j);
                intPt.x = line.pt2.x;
                intPt.y = line.pt2.y;
                m_point_Array.Add(intPt);
            }
        }
    }
    else{//>0 多边形初始是逆时针
        if(ringFlag == 0){
            //是外环,顶点序列按初始边的反方向顺序排序
            for(int j = m_PolyLine_array.GetSize() - 1;j >= 0;j -- )  {
                line = m_PolyLine_array.GetAt(j);
```

```
                    intPt.x = line.pt2.x;
                    intPt.y = line.pt2.y;
                    m_point_Array.Add(intPt);
                }
            }
            else {
                //是内环,顶点序列按初始边的顺序排序
                for(int j = 0;j < m_PolyLine_array.GetSize();j++){
                    line = m_PolyLine_array.GetAt(j);
                    //m_point_Array.Add(line.pt1);
                    intPt.x = line.pt1.x;
                    intPt.y = line.pt1.y;
                    m_point_Array.Add(intPt);
                }
            }
        }
        m_point_Array.Add(m_point_Array.GetAt(0));        //使首尾相接
    }
```

其中,在判断裁剪多边形的走向时,需要计算有向线段的叉积,函数代码为:

```
/* pt0,pt2,pt2 是顺序三个顶点 */
int VectorXVector(CPoint &pt0,CPoint &pt1,CPoint &pt2){
    return (pt1.x - pt0.x) * (pt2.y - pt1.y) - (pt2.x - pt1.x) * (pt1.y - pt0.y);
}
```

计算两条直线交点的函数代码为:

```
/* 计算直线段 pts - pte 和直线段 pt0 - pt1 的交点 InterPt  */
void InterPtToPt(CPoint &pts,CPoint &pte,CPoint &pt0,CPoint &pt1,CPoint &InterPt){
    int a1,b1,c1,a2,b2,c2;
    a1 = pts.y - pte.y;
    b1 = pte.x - pts.x;
    c1 = pts.x * pte.y - pte.x * pts.y;
    a2 = pt0.y - pt1.y;
    b2 = pt1.x - pt0.x;
    c2 = pt0.x * pt1.y - pt1.x * pt0.y;
    InterPt.x = int(0.5 + (double)(c2 * b1 - c1 * b2)/(double)(a1 * b2 - a2 * b1));
    InterPt.y = int(0.5 + (double)(c1 * a2 - c2 * a1)/(double)(a1 * b2 - a2 * b1));
}
```

图 4.2-9 所示为利用上述代码实现的任意多边形被一个凸多边形裁剪以及区域填充的效果。

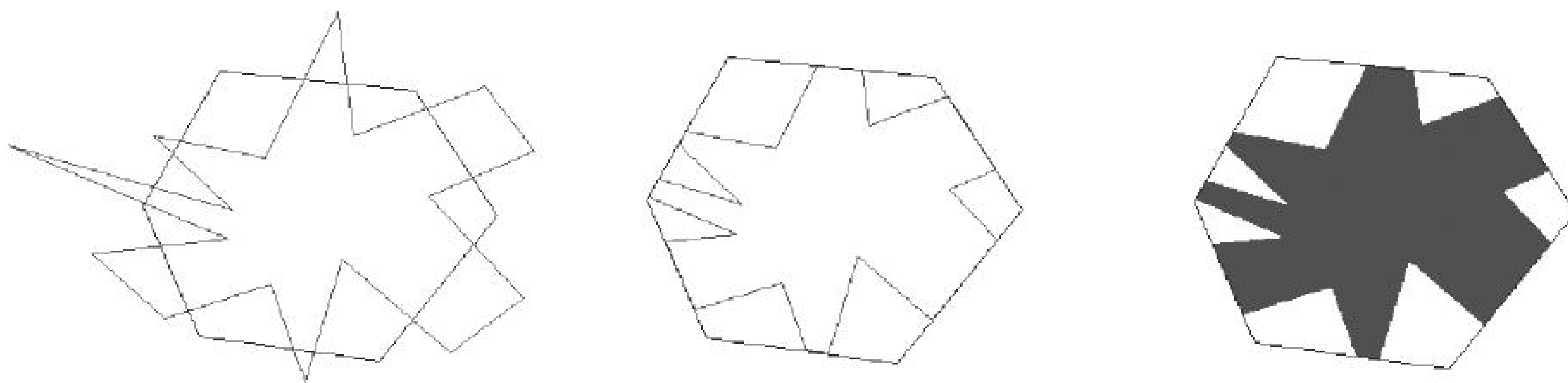


图 4.2-9　一般凸多边形裁剪窗口裁剪多边形

逐边裁剪法的特点是理论简单，也比较容易实现，但是它主要适合于裁剪窗口是凸多边形的情况。当裁剪窗口是任意的形状，例如凹多边形和带内环的多边形时，利用裁剪边界来判断多边形顶点在裁剪窗口哪一侧就会失效。而且被裁剪多边形是凹多边形时，利用逐边裁剪法有时会出现“退化边”的情况。如图 4.2-10 所示的凹多边形利用逐边裁剪法裁剪后，会形成有边界重合的一个多边形，看似却是多个多边形，而且之间有连线，此即为“退化边”现象。

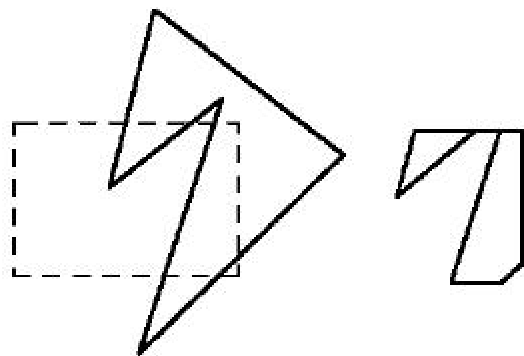


图 4.2-10　逐边裁剪法的退化边现象

4.2.3　任意形状多边形的裁剪

Sutherland-Hodgman 逐边裁剪算法解决了裁剪窗口为凸多边形的多边形裁剪问题，当裁剪窗口是任意形状的多边形时，例如凹多边形以及更一般的带内环的多边形时，Weiler-Atherton 多边形裁剪算法具有更好的解决思路。由于 Weiler-Atherton 裁剪算法中被裁剪多边形和裁剪多边形相互裁剪，故该算法又称为双边裁剪算法(简称 W-A 算法)，另外 Weiler-Atherton 裁剪算法还可以有效地解决 S-H 算法中的“退化边”问题。

在 Weiler-Atherton 裁剪算法中，裁剪窗口和被裁剪多边形处于完全对等的地位。其中，为了区分，被裁剪多边形即为主多边形，记为 SP；裁剪多边形即为裁剪窗口，记为 CP，如图 4.2-11 所示。

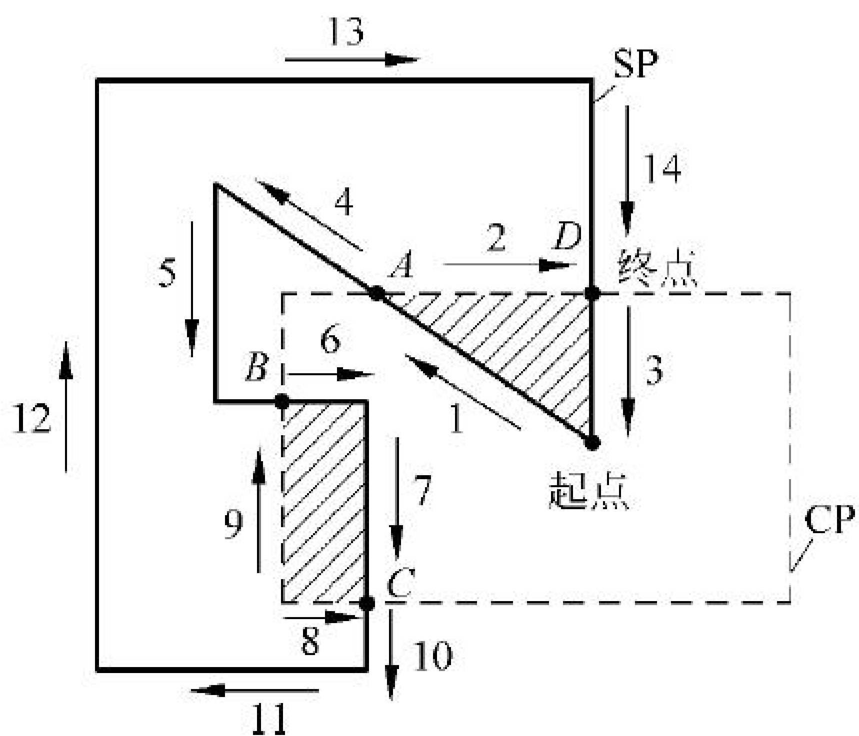


图 4.2-11　被裁剪多边形 SP 和裁剪多边形 CP

从图 4.2-11 中可以看出，多边形裁剪的结果是裁剪后的多边形边界由被裁剪多边形 SP 和裁剪多边形 CP 两部分的边界组成，并且边界在两个多边形的交点处发生交替，即由被裁剪多边形 SP 的边界转至裁剪多边形 CP 的边界，或者由裁剪多边形 CP 的边界转至被裁剪多边形 SP 的边界。由于多边形构成的是一个封闭的区域，所以，如果被裁剪多边形和裁剪多边形有交点，则交点成对出现。这些交点分为两类。

(1) 进点。即被裁剪多边形由此点进入裁剪多边形，如图中的点 B、D。

(2) 出点。即被裁剪多边形由此点离开裁剪多边形，如图中的点 A、C。

Weiler-Atherton 裁剪算法的思想：从被裁剪多边形的一个顶点开始，碰到进点，沿着被裁剪多边形按顺时针方向搜集顶点序列；而当遇到出点时，则沿着裁剪多边形按顺时针方向搜集顶点序列，当遇到进点时，再按照被裁剪多边形顺时针方向搜集顶点序列。按上述规则，如此交替地沿着两个多边形的边线行进，直到回到起始进点。这时，收集到的全部顶点序列就是裁剪所得的一个多边形。

特别需要注意的是，由于可能被裁剪成多个分离的多边形，如图 4.2-11 所示的情况，那

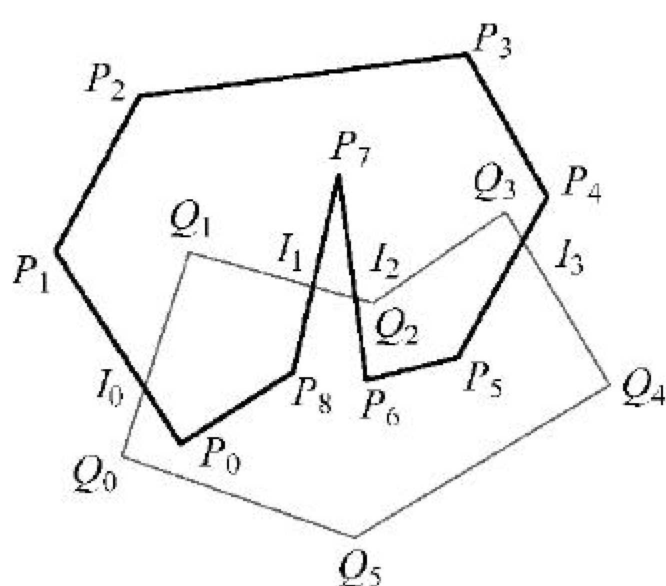


图 4.2-12 Weiler-Atherton 双边裁剪算法

么，在沿着裁剪多边形顺时针搜集顶点时，如遇到进点，需要判断该进点是否已搜集过，如该进点已经搜集过，则现有搜集到的顶点序列已经形成一个封闭的多边形，本次搜集结束。然后，从被裁剪多边形的下一个未搜集的进点出发重新开始搜集顶点序列。循环重复上述步骤，直至将所有的进点搜集完毕后算法结束。

如图 4.2-12 所示的被裁剪多边形 $P_0P_1P_2P_3P_4P_5P_6P_7P_8$ 及裁剪多边形 $Q_0Q_1Q_2Q_3Q_4Q_5$，两个多边形的顶点均已按顺时针方向进行排序，两个多边形的交点为 I_0、I_1、I_2、I_3，其中，I_0、I_2 为出点，I_1、I_3 为进点。然后，将交点分别插入两个多边形的顶点序列中：

插入交点后的被裁剪多边形顶点序列为 $P_0I_0P_1P_2P_3P_4I_3P_5P_6I_2P_7I_1P_8$

插入交点后的裁剪多边形顶点序列为 $Q_0I_0Q_1I_1I_2Q_2Q_3I_3Q_4Q_5$

从被裁剪多边形的顶点序列中找到一个进点 I_3，并沿着被裁剪多边形搜集顶点序列——P_5、P_6，一直到交点 I_2，I_2 为出点，则从 I_2 点开始沿裁剪多边形方向搜集顶点序列——Q_2、Q_3、I_3，由于 I_3 为本次搜集的初始进点，则这次顶点搜集结束，并形成一个裁剪后的多边形——$I_3P_5P_6I_2Q_2Q_3$。

接着再查找被裁剪多边形下一个进点 I_1，并沿着被裁剪多边形搜集顶点序列——P_8、P_0，到交点 I_0，I_0 为出点，则从 I_0 点开始沿裁剪多边形搜集顶点——Q_1、I_1，由于 I_1 为本次搜集的初始进点，则本次搜集结束，形成的多边形为 $I_1P_8P_0I_0Q_1$。由于不再有未搜集的进点，则裁剪结束。

在用代码实现 Weiler-Atherton 裁剪算法时，有两个关键点：一是裁剪多边形和被裁剪多边形的顶点都需要进行正确排序，其中外环按照顺时针排序，内环按照逆时针排序；二是计算两个相互裁剪的多边形交点，并将交点正确插入到多边形的顶点序列中。

对于凸多边形的顶点排序，在 4.2.2 节中曾经提出了有向线段的向量叉积判别法进行判断排序，该方法主要适用于裁剪多边形是凸多边形的情况，当多边形是其他形状，例如凹多边形和带内环的多边形时，需要对该方法做进一步的改进才能正确判断多边形走向，例如，利用多边形的极值边的有向线段的向量叉积来判断。本节提出了另外一种比较实用的新的判别方法：利用多边形扫描转换中扫描线算法的原理来判断多边形边的右侧区域的点是否在多边形内部，从而判别和设置多边形的顶点走向(该法可称为基于扫描线算法的多边形方向判别法)。

如图 4.2-13 所示，对于多边形 $P_0P_1P_2P_3P_4P_5P_6$，假设一个人沿着多边形的边界线前进，在外环沿顺时针方向前进，在内环沿逆时针方向前进，此时，其身体右侧的点始终是多边形的内部点。据此，可以判断多边形的方向：沿着多边形走向前进时，如果身体右侧的点在多边形内部，则多边形为顺时针走向，否则多边形为逆时针走向。利用这个方法进行判断时，首先需要沿着多边形的走向构造一个在某条边右侧的点，接

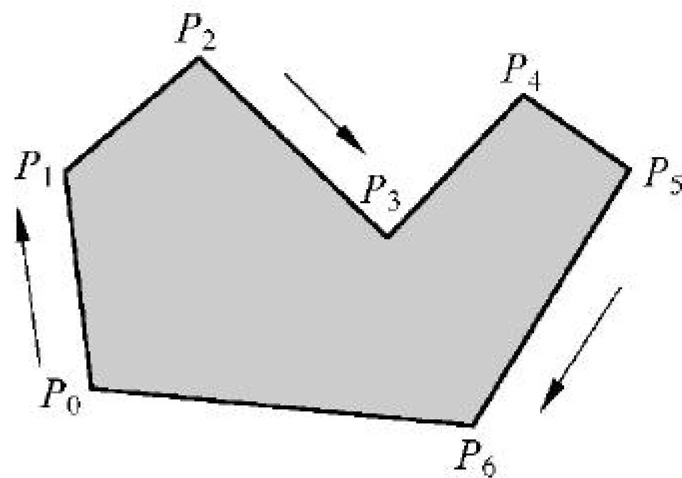


图 4.2-13 多边形内部填充色判别法

着判断该右侧点是否在多边形内部。

假设多边形外环是顺时针方向，即边 P_0P_1 方向为 $P_0(x_0,y_0)\rightarrow P_1(x_1,y_1)$，边 P_0P_1 上的点为 $\begin{cases}x=(1-t)x_0+tx_1\\y=(1-t)y_0+ty_1\end{cases}$，$0\leqslant t\leqslant 1$，由于边线段会有不同的倾斜方向，因此该边右侧的点会有以下四种情况，如图 4.2-14 所示。

设多边形边上的点为 (x,y)，右侧点 (x_i,y_i) 可以用单位增量点表示，当 $x_0\leqslant x_1,y_0\leqslant y_1$ 时，即图 4.2-14 中的第一种情况，当用点阵表示直线时，右侧单位增量点可以是 $(x+1,y)$，$(x,y-1)$ 和 $(x+1,y-1)$，如图 4.2-15 所示，右侧点取 $(x+1,y-1)$。同理，边直线其他几种倾斜情况下的右侧点取值分别为

当 $x_0\leqslant x_1,y_0\geqslant y_1$ 时，$x_i=x-1,y_i=y-1$；

当 $x_0\geqslant x_1,y_0\leqslant y_1$ 时，$x_i=x+1,y_i=y+1$；

当 $x_0\geqslant x_1,y_0\geqslant y_1$ 时，$x_i=x-1,y_i=y-1$。

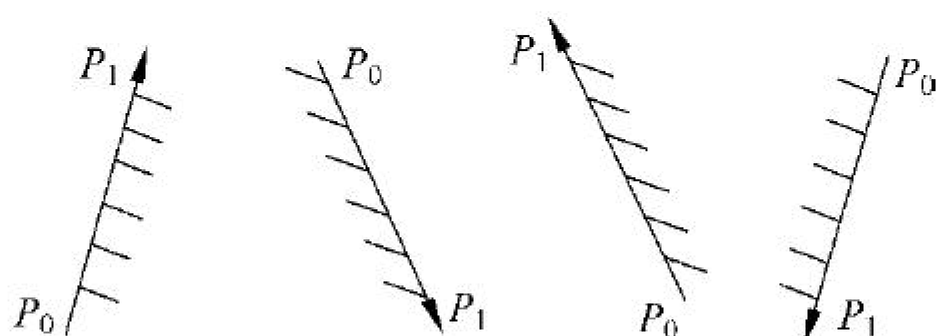


图 4.2-14　多边形边右侧点的四种情况

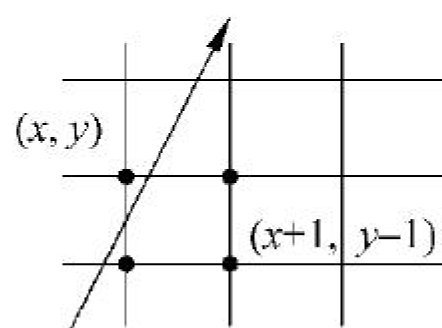


图 4.2-15　直线右侧点

判断右侧点是否在多边形内部，最直观的方法是借助多边形扫描转换中的扫描线算法原理：利用扫描线 $y=y_i$ 与多边形各边求交点，顺序组成交点区间对，如果右侧点在某一交点区间对中，则右侧点在多边形内部，多边形是顺时针走向，否则是逆时针走向，如图 4.2-16 所示。

Weiler-Atherton 裁剪算法的第二个关键点是两个多边形交点的计算。和逐边裁剪法中多边形和裁剪边界的交点计算方法不同，两个多边形的交点必须在边的两个端点之间，不能在边的延长线上，否则为无效交点。为此，可借助直线段的参数方程进行求解。设被裁剪多边形某边的两个顶点是 $P_s(x_s,y_s)$，$P_e(x_e,y_e)$，裁剪多边形边的两个顶点是 $P_a(x_a,y_a)$，$P_b(x_b,y_b)$，则两个边的参数式方程分别为

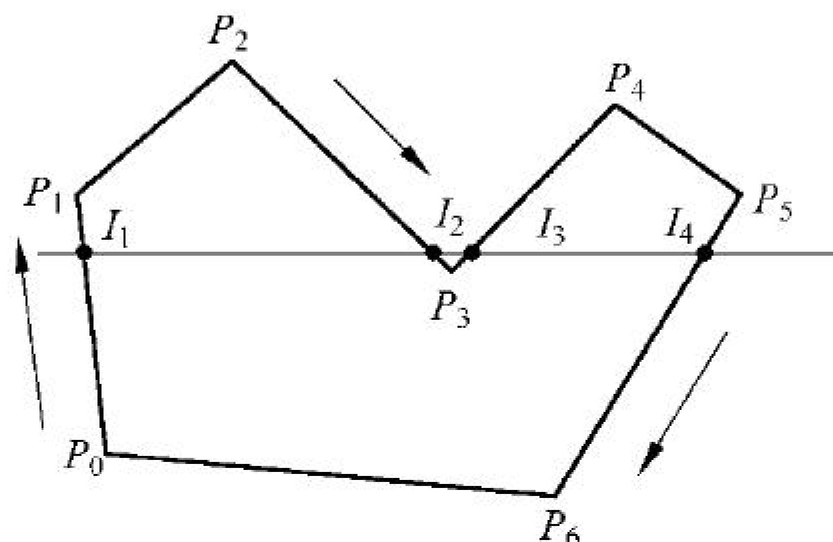


图 4.2-16　扫描线法判断多边形内部点

$$\begin{cases}x=(1-u)x_s+ux_e\\y=(1-u)y_s+uy_e\end{cases},\quad 0\leqslant u\leqslant 1$$

和

$$\begin{cases}x=(1-v)x_a+vx_b\\y=(1-v)y_a+vy_b\end{cases},\quad 0\leqslant v\leqslant 1$$

当两个边相交即有共同点时，联立解方程

$$\begin{cases}(1-u)x_s+ux_e=(1-v)x_a+vx_b \\ (1-u)y_s+uy_e=(1-v)y_a+vy_b\end{cases},\quad 0\leqslant u\leqslant 1,0\leqslant v\leqslant 1$$

得

$$\begin{cases}u=\dfrac{(x_a-x_s)(y_b-y_a)-(y_a-y_s)(x_b-x_a)}{(x_e-x_s)(y_b-y_a)-(y_e-y_s)(x_b-x_a)} \\ v=\dfrac{(x_a-x_s)(y_e-y_s)-(y_a-y_s)(x_e-x_s)}{(x_e-x_s)(y_b-y_a)-(y_e-y_s)(x_b-x_a)}\end{cases}$$

当所得的结果同时满足 $0\leqslant u<1$ 和 $0\leqslant v<1$ 时(注意边界一端是开的),两个边有交点,否则没有交点。

当一条边有多个交点时,需要按照每个交点对应的参数值 u 或者 v 的大小顺序,将交点插入多边形顶点序列中。

在对多边形顶点和交点一起排序时,每个点的信息中不仅要记录坐标信息,还需要记录该点是否为相交点,以及是进点还是出点。由于一条边上可能会有多个交点,需要对这些交点排序。为了实现排序比较,点信息中还需记录每个点在边上的参数值 u 和 v 的大小,故多边形交点的结构类设计如下:

```
class CInterPoint{                //多边形交点类
public:
    int x;
    int y;
    int flag;                     //交点标识,0: 非交点,1: 进点,-1: 出点,2: 该进点已经使用
    double parau;                 //顶点是交点时,在被裁剪多边形对应边的参数方程的参数值
    double parav;                 //顶点是交点时,在裁剪多边形对应边的参数方程的参数值
public:
    CInterPoint(){
        flag = 0;
        parau = 0;
        parav = 0;
    }
};
```

Weiler-Atherton 裁剪算法的函数参考代码如下:

```
/**********************************************************************
Polyline_CutToWA: Weiler-Atherton 裁剪算法函数
m_PolyLine_array: 任意多边形; m_cutPolyLine: 任意多边形裁剪窗口; m_MorePolyLine: 裁剪后另
外生成的多边形数组; morePolyNum: 另外生成的数目
**********************************************************************/
void Polyline_CutToWA(CArray<CLine,CLine> &m_PolyLine_array, int ringFlag, CPolyLine &m_
cutPolyLine, CPolyLine * m_MorePolyLine, int &morePolyNum){
    //ringFlag = 0: 外环,1: 内环
    CArray<CInterPoint,CInterPoint> m_point_Array,m_cut_point_Array;  //原始排序后的顶点序列
    CArray<CInterPoint,CInterPoint> m_new_cut_point_Array;  //加入交点后的裁剪多边形顶
//点序列,被裁剪多边形还用原来的顶点序列加交点
    CArray<CPoint,CPoint> m_point_Array1;                 //裁剪后的顶点序列
    //1.设置裁剪多边形走向
    SortForPolyline(m_PolyLine_array,ringFlag,m_point_Array);
```

```
    //2.设置裁剪多边形走向
    SortForPolyline(m_cutPolyLine.m_PolyLine_array_Out,0,m_cut_point_Array);
    //3.循环从裁剪多边形中逐次取出一条边,被裁剪多边形每条边循环判断相交,并求交点,将交
//点加入两个多边形的顶点序列中
    CInterPoint ptA,ptB,pt_0,pt_1,interPt;
    CArray<CInterPoint,CInterPoint> m_InterPt_Array;    //记录当前边相交点的集合
    int inOroutflag = 0;                                  //0: 外侧,1: 内侧
    for(int j = 0;j < m_cut_point_Array.GetSize() - 1;j++){
        ptA = m_cut_point_Array.GetAt(j);
        m_new_cut_point_Array.Add(ptA);                   //将每条边的第一个顶点首先插入
        ptB = m_cut_point_Array.GetAt(j + 1);
        //取多边形第一个顶点,判断在内侧还是外侧
        pt_0 = m_point_Array.GetAt(0);
        if(VectorXVector(ptB,ptA,pt_0) * ( - 1)> 0)       //外侧
            inOroutflag = 0;
        else
            inOroutflag = 1;                              //内侧
        //循环从被裁剪多边形中取出一个顶点,判断是否有交点,如有交点,则加入
        m_InterPt_Array.RemoveAll();                      //清空当前裁剪边的交点序列
        for(int k = 1;k < m_point_Array.GetSize();k++){
            pt_1 = m_point_Array.GetAt(k);
            //取顶点判断在内侧还是外侧
            if(VectorXVector(ptB,ptA,pt_1) * ( - 1)> 0){  //外侧
                if(inOroutflag == 0) {                    //如果上一个顶点也在外侧
                    pt_0 = pt_1;
                    continue; }
                else {                                    //上一个顶点在内侧
                    //计算交点
                    if(InterPtToPt(ptA,ptB,pt_0,pt_1,interPt) == 1)  {
                        interPt.flag =- 1;                //有交点,设置该交点是出点标识
                        //在裁剪多边形顶点的合适位置插入交点
        InsertPtInCutPts(interPt,m_new_cut_point_Array,m_InterPt_Array);
                        //被裁剪多边形顶点序列中加入交点
                        m_point_Array.InsertAt(k,interPt);
                        k++;                              //下一次从交点的下一个顶点开始
                    }
                    inOroutflag = 0;
                    pt_0 = pt_1;
                }
            }
            else{                                         //内侧
                if(inOroutflag == 1){                     //也在内侧
                    pt_0 = pt_1;
                    continue;
                }
                else  {
                    //上一个顶点在外侧,则有交点
                    if(InterPtToPt(ptA,ptB,pt_0,pt_1,interPt) == 1){
                        interPt.flag = 1;                 //有交点,设置该交点是进点标识
                        //在裁剪多边形顶点的合适位置插入交点
        InsertPtInCutPts(interPt,m_new_cut_point_Array,m_InterPt_Array);
```

```
                    //被裁剪多边形顶点序列中加入交点
                    m_point_Array.InsertAt(k,interPt);
                    k++;                            //下一次从交点的下一个顶点开始
                }
                inOroutflag = 1;                    //设置新的 inOroutflag = 1 在内侧
                pt_0 = pt_1;
            }
        }
    }
}
m_new_cut_point_Array.Add(m_new_cut_point_Array.GetAt(0));
```

/*循环从被裁剪多边形的顶点序列中查找进点,然后,从被裁剪多边形搜集顶点,遇到出点,则从裁剪多边形搜集顶点,直到遇到进点。如果进点是初始查找到的进点,则此轮搜集结束,形成一个多边形,将该进点的标识设为非进点,重新从下一个进点开始搜集*/

```
morePolyNum =- 1;                   //额外形成的多边形的数量, = 0 正在形成原初始多边形
m_point_Array1.RemoveAll();
int indexC;
CPoint pt0;
CLine line;
CInterPoint tPt;
j = 0;
int Hflag = 0;                                  //是否有进口的标识
while(1)  {
    if(m_point_Array.GetAt(j).flag == 1){
        Hflag = 1;
        //进口,加入顶点序列
        pt0.x = m_point_Array.GetAt(j).x;
        pt0.y = m_point_Array.GetAt(j).y;
        //把该点的进点标识去掉
        tPt = m_point_Array.GetAt(j);
        tPt.flag = 2;
        m_point_Array.RemoveAt(j);
        m_point_Array.InsertAt(j,tPt);
        m_point_Array1.Add(pt0);
        while(1)  {
            j++;                                //沿被裁剪多边形搜集多边形顶点
            if(j >= m_point_Array.GetSize()) j = 0;
            pt0.x = m_point_Array.GetAt(j).x;
            pt0.y = m_point_Array.GetAt(j).y;
            m_point_Array1.Add(pt0);
            //判断是否为出点
            if(m_point_Array.GetAt(j).flag ==- 1){
                //是出点,在裁剪多边形中查找位置,并从该位置开始搜集顶点
    indexC = FindPolylinePt(m_new_cut_point_Array,m_point_Array.GetAt(j));
                if(indexC!=- 1){
                    int iflag = 0;
                    while(1){                   //沿裁剪多边形顶点序列搜集
                        indexC++;
if(indexC >= m_new_cut_point_Array.GetSize())indexC = 0;   //从初始点开始
                        pt0.x = m_new_cut_point_Array.GetAt(indexC).x;
                        pt0.y = m_new_cut_point_Array.GetAt(indexC).y;
```

```
                                m_point_Array1.Add(pt0);
                                //判断是否为进点
                                if(m_new_cut_point_Array.GetAt(indexC).flag == 1){
/*是进点,判断是否为本次搜集的初始进点,如是,则形成一个多边形;如不是,则返回被裁剪多边
形继续搜集*/
    if(pt0.x == m_point_Array1.GetAt(0).x&&pt0.y == m_point_Array1.GetAt(0).y){
                                        //形成一个封闭多边形,记录该多边形
                                        if(morePolyNum ==- 1){
                                            //形成原初始多边形
                                            m_PolyLine_array.RemoveAll();
                                for(int jj = 0;jj < m_point_Array1.GetSize() - 1;jj++){
                                            line.pt1 = m_point_Array1.GetAt(jj);
                                            line.pt2 = m_point_Array1.GetAt(jj + 1);
                                            m_PolyLine_array.Add(line);
                                            }
                                        }
                                        else  {             //另外形成的多边形
                                for(int jj = 0;jj < m_point_Array1.GetSize() - 1;jj++){
                                        line.pt1 = m_point_Array1.GetAt(jj);
                                        line.pt2 = m_point_Array1.GetAt(jj + 1);
            m_MorePolyLine[morePolyNum].m_PolyLine_array_Out.Add(line);}}
                                        morePolyNum++;
                                        m_point_Array1.RemoveAll();
                                        iflag =- 1;
                                        break;
                                    }
                                    else{
                                //新进点,则查找新进点在被裁剪多边形的位置
indexC = FindPolylinePt(m_point_Array,m_new_cut_point_Array.GetAt(indexC));
                                        iflag = 1;
                                        j = indexC;
                                        break;
                                    }
                                }
                            }
                            if(iflag == 1)
                                continue;          //被裁剪多边形从 j 点开始继续搜集
                            else if(iflag ==- 1)
                                break;             //本次多边形已经完成,重新从 j 点开始搜集
                            }
                    }
                }
            }
            else if(j == m_point_Array.GetSize() - 1&&Hflag == 1)  {
                j = 0;                              //从头开始再判断是否还有进口
                Hflag = 0;
                continue;
            }
            else if(j == m_point_Array.GetSize() - 1&&Hflag == 0)
                break;                              //不再有进口,则结束搜集
            else{
```

```
                j++;
                continue;
            }
        }
    }
```

其中,多边形方向的排序函数代码为:

```
/*m_PolyLine_array:多边形;ringFlag:多边形方向;m_point_Array:排序后的顶点序列*/
void SortForPolyline(CArray<CLine,CLine>&m_PolyLine_array,int ringFlag,CArray
<CInterPoint,CInterPoint>&m_point_Array){
    //判断多边形外环是否顺时针方向,如不是,把顶点序列改为正确顺序,外环顺时针,内环逆时针
    CPolyLine pL;
    pL.m_PolyLine_array_Out.Append(m_PolyLine_array);
    CLine line;
    CInterPoint intPt;
    if(CheckDirectOfPolyline(pL)==0){            //顺时针
        if(ringFlag==0){                          //是外环,顶点序列按初始边的顺序排序
            for(int j=0;j<m_PolyLine_array.GetSize();j++){
                line=m_PolyLine_array.GetAt(j);
                intPt.x=line.pt1.x;
                intPt.y=line.pt1.y;
                m_point_Array.Add(intPt);
            }
        }
        else{                                     //是内环,顶点序列按初始边的反方向顺序排序
            for(int j=m_PolyLine_array.GetSize()-1;j>=0;j--){
                line=m_PolyLine_array.GetAt(j);
                intPt.x=line.pt2.x;
                intPt.y=line.pt2.y;
                m_point_Array.Add(intPt);
            }
        }
    }
    else{                                         //>0 多边形初始是逆时针
        if(ringFlag==0){
            //是外环,顶点序列按初始边的反方向顺序排序
            for(int j=m_PolyLine_array.GetSize()-1;j>=0;j--)  {
                line=m_PolyLine_array.GetAt(j);
                intPt.x=line.pt2.x;
                intPt.y=line.pt2.y;
                m_point_Array.Add(intPt);
            }
        }
        else{                                     //是内环,顶点序列按初始边的顺序排序
            for(int j=0;j<m_PolyLine_array.GetSize();j++)  {
                line=m_PolyLine_array.GetAt(j);
                //m_point_Array.Add(line.pt1);
                intPt.x=line.pt1.x;
                intPt.y=line.pt1.y;
                m_point_Array.Add(intPt);
```

```
            }
        }
    }
    m_point_Array.Add(m_point_Array.GetAt(0)); //首尾相接
}
```

利用扫描线原理判断多边形走向的函数代码为：

```
int CheckDirectOfPolyline(CPolyLine &polyline){
    CPoint pt0,pt1,pt_t;
    int k = 0;
    while(1)  {
        pt0 = polyline.m_PolyLine_array_Out.GetAt(k).pt1;
        pt1 = polyline.m_PolyLine_array_Out.GetAt(k).pt2;
        if(pt0.y == pt1.y) k++;
        else
            break;                          //避免水平线
    }
    //随机判断5次(最好接近中部),如是,是顺时针; 否,是逆时针
    double t;
    int iYesFlag = 0;
    int iNoFlag = 0;
    int iFlag = 0;
    while(1)  {
        t = (rand() % 100)/100.0;           //产生随机数
        if(t <= 0.1||t > 0.9)               //避免靠近端点
            continue;
        pt_t.x = (int)(pt0.x * (1 - t) + t * pt1.x + 0.5);
        pt_t.y = (int)(pt0.y * (1 - t) + t * pt1.y + 0.5);
        if(pt0.x <= pt1.x) {                //确定一个右侧点
            if(pt0.y <= pt1.y) {
                pt_t.x += 1;
                pt_t.y -= 1;
            }
            else  {
                pt_t.x -= 1;
                pt_t.y -= 1;
            }
        }
        else  {
            if(pt0.y <= pt1.y) {
                pt_t.x += 1;
                pt_t.y += 1;
            }
            else  {
                pt_t.x -= 1;
                pt_t.y += 1;
            }
        }
        //判断扫描线和边是否相交,如相交,求交点,并排序
        CArray < int, int > m_x_Array;
```

```
        int m_x;                                    //交点
        int j,k;
        m_x_Array.RemoveAll();
        //首先判断扫描线和外环多边形的交点
        for(int i = 0;i < polyline.m_PolyLine_array_Out.GetSize();i++)  {
        //将每条边的最大 y 值缩短一个单位,判断是否和扫描线相交,如相交,求交点,插入交点
//集并排序
if((pt_t.y >= polyline.m_PolyLine_array_Out.GetAt(i).pt1.y&&pt_t.y < polyline.m_PolyLine_
array_Out.GetAt(i).pt2.y)||(pt_t.y >= polyline.m_PolyLine_array_Out.GetAt(i).pt2.y&&pt_t.y
< polyline.m_PolyLine_array_Out.GetAt(i).pt1.y)){  //求交点
m_x = GetInterPtXForScanY(pt_t.y,polyline.m_PolyLine_array_Out.GetAt(i).pt1.x,polyline.m_
PolyLine_array_Out.GetAt(i).pt1.y,polyline.m_PolyLine_array_Out.GetAt(i).pt2.x,polyline.m
_PolyLine_array_Out.GetAt(i).pt2.y);            //排序
                    OrderToInsertPt_x(m_x_Array,m_x);
                }
                else                                //水平线时,重新找交点
                    continue;
            }
            //再判断扫描线和内环多边形的交点
            CArray < CLine,CLine > m_line_Array_in;
            for(k = 0;k < polyline.in_num;k++)  {
        m_line_Array_in.Append(polyline.m_PolyLine_array_in[k]);  //内环多边形
                for(i = 0;i < m_line_Array_in.GetSize();i++)  {
                    //将每条边的最大 y 值缩短一个单位,判断是否和扫描线相交,如相交,求交
点,插入交点集并排序
if((pt_t.y >= m_line_Array_in.GetAt(i).pt1.y&&pt_t.y < m_line_Array_in.GetAt(i).pt2.y)||
(pt_t.y >= m_line_Array_in.GetAt(i).pt2.y&&pt_t.y < m_line_Array_in.GetAt(i).pt1.y))  {
                        //求交点
m_x = GetInterPtXForScanY(pt_t.y,m_line_Array_in.GetAt(i).pt1.x,m_line_Array_in.GetAt(i).
pt1.y,m_line_Array_in.GetAt(i).pt2.x,m_line_Array_in.GetAt(i).pt2.y);
                        //排序
                        OrderToInsertPt_x(m_x_Array,m_x);
                    }
                    else//水平线
                        continue;
                }
                m_line_Array_in.RemoveAll();
            }
            //判断是否在区间对
            iFlag = 0;
            for(j = 0;j <= m_x_Array.GetSize() - 2;j++,j++){
                if(pt_t.x >= m_x_Array.GetAt(j)&&pt_t.x <= m_x_Array.GetAt(j + 1))  {
                    iYesFlag++;
                    iFlag = 1;
                    break;
                }
        }
        if(iFlag == 0)
            iNoFlag++;
        if(iYesFlag > 5)
            break;
```

```
        else if(iNoFlag > 5)
            break;
    }
    if(iYesFlag > 5)  return 0;                  //顺时针 1
    else return 1;                               //逆时针
}
```

上述扫描线法判断多边形走向的函数中使用的计算扫描线和边的交点的函数 GetInterPtXForScanY()和排序的函数 OrderToInsertPt_x()请参考 3.4.1 节中相关代码。其中,计算两个直线段交点的函数代码如下:

```
/* 计算直线段 pts - pte 和直线段 pt0 - pt1 的交点 InterPt */
int InterPtToPt ( CInterPoint &pts, CInterPoint &pte, CInterPoint &pta, CInterPoint &ptb,
CInterPoint &InterPt){
    InterPt.parau = (double)((pta.x - pts.x) * (pte.y - pts.y) - (pta.y - pts.y) * (pte.x - pts.
x))/(double)((pte.x - pts.x) * (ptb.y - pta.y) - (pte.y - pts.y ) * (ptb.x - pta.x));
    InterPt.parav = (double)((pta.x - pts.x) * (ptb.y - pta.y) - (pta.y - pts.y) * (ptb.x - pta.
x))/(double)((pte.x - pts.x) * (ptb.y - pta.y) - (pte.y - pts.y ) * (ptb.x - pta.x));
    if((InterPt.parau >= 0&&InterPt.parau < 1)&(InterPt.parav >= 0&&InterPt.parav < 1)){
        //有交点(注意一端是封闭的,另一端是开口的)
        InterPt.x = (int)(0.5 + (1 - InterPt.parav) * pts.x + InterPt.parav * pte.x);
        InterPt.y = (int)(0.5 + (1 - InterPt.parav) * pts.y + InterPt.parav * pte.y);
        return 01;
    }
    else                                         //没有交点
        return 0;
}
```

在裁剪多边形中插入交点的函数代码为:

```
/* interPt : 交点; m_new_cut_point_Array: 裁剪多边形顶点序列; m_InterPt_Array: 当前边上的
交点集合,用于判断交点插入位置 */
void InsertPtInCutPts(CInterPoint &interPt, CArray < CInterPoint, CInterPoint > &m_new_cut_
point_Array,CArray < CInterPoint,CInterPoint > &m_InterPt_Array){
    //判断该交点在裁剪边上添加的位置
    CInterPoint nPt;                             //交点序列中的交点
    int nFlag = 0;                               //交点参数判断标识符
    if(m_InterPt_Array.GetSize()> 0){
        nFlag = 0;                               //设置初始为 0
        for(int m = 0;m < m_InterPt_Array.GetSize();m++){
            nPt = m_InterPt_Array.GetAt(m);
            if(nPt.parav < interPt.parav)
                continue;
            else  {
                //比交点集合中的参数值小,则插入之前
m_new_cut_point_Array.InsertAt(m_new_cut_point_Array.GetSize() - m_InterPt_Array.GetSize()
 + m,interPt);                                       //裁剪多边形顶点序列中加入交点
                m_InterPt_Array.InsertAt(m,interPt);
                nFlag = 1;                           //设置为 1
```

```
                break;
            }
        }
        if(nFlag == 0){                         //没有插入,则插入最后
            m_InterPt_Array.Add(interPt);       //交点序列最后加入
            m_new_cut_point_Array.Add(interPt); //裁剪多边形顶点序列最后加入
        }
    }
    else  {
        m_InterPt_Array.Add(interPt);           //交点序列最后加入
        m_new_cut_point_Array.Add(interPt);     //裁剪多边形顶点序列最后加入交点
    }
}
```

查找点在多边形顶点序列位置的函数代码如下：

```
/*m_point_Array: 多边形顶点序列; InterPt: 查找位置的顶点*/
int FindPolylinePt(CArray<CInterPoint,CInterPoint> &m_point_Array,CInterPoint &InterPt){
//在裁剪多边形顶点序列查找点的位置
    int i =- 1;
    int flag = 0;
    CInterPoint pt;
    for(i = 0;i < m_point_Array.GetSize();i++){
        pt = m_point_Array.GetAt(i);
        if(pt.x == InterPt.x&&pt.y == InterPt.y)  {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        return i;
    else
        return -1;
}
```

利用上述的Weiler-Atherton裁剪算法,针对任意形状的裁剪多边形窗口和被裁剪多边形均可实现正确裁剪。图4.2-17所示为利用上述的算法代码对任意多边形裁剪前后的填充效果,图4.2-18所示为裁剪后会产生多个多边形的情况,从图4.2-18中也可以看出Weiler-Atherton裁剪算法能够避免出现退化边的情况。

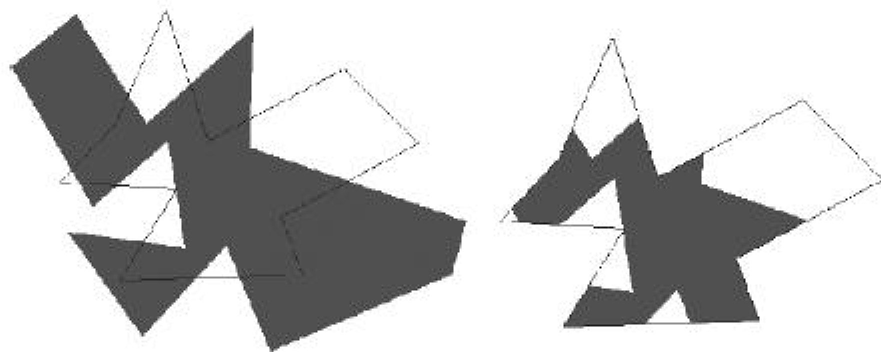


图4.2-17 多边形裁剪

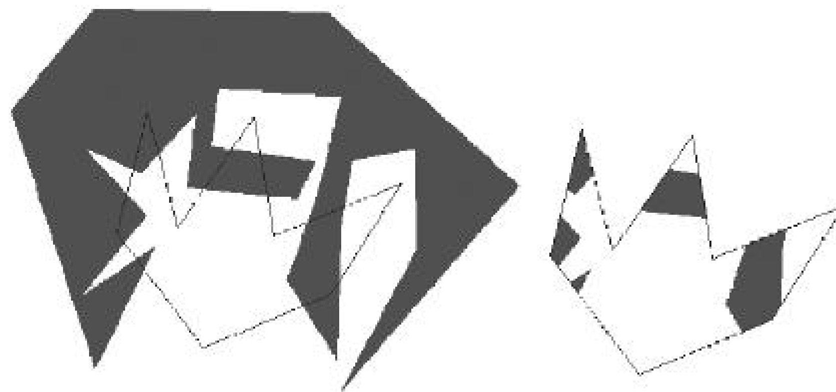


图4.2-18 避免裁剪退化边现象

4.3 圆 裁 剪

4.3.1 圆裁剪概述

圆裁剪有两种类型：一是圆作为裁剪窗口去裁剪其他图形；二是圆作为被裁剪图形，被其他裁剪窗口裁剪。在第一种圆裁剪类型中，圆作为裁剪窗口，对其他图形进行裁剪，仅保留圆形内部的部分，其他部分都被裁剪掉。第二种圆裁剪类型是指一个整圆或者一段圆弧被指定的一个裁剪窗口进行裁剪，裁剪后只保留窗口内部的圆弧部分，窗口外的圆弧部分被裁剪掉。两种类型的圆裁剪在工程上都有实用价值。

在圆裁剪中，为了使算法能够实现，有时也需要类似多边形方向的设定方式，对圆按照顺时针或者逆时针走向进行标识，把顺时针方向设为圆的正向，逆时针方向作为负向；反之也可以。

圆作为被裁剪对象被窗口进行裁剪后，结果将不再是整圆，而是圆弧段。圆弧段是圆的一部分，也可以被窗口裁剪，因此，圆裁剪不仅限于整圆被裁剪，一段圆弧也可以被裁剪。

4.3.2 圆形窗口的线段裁剪

圆形窗口的线段裁剪是指圆作为裁剪窗口裁剪线段，只保留圆内部的线段部分。圆和线段的位置关系有以下四种情况，如图 4.3-1 所示。

（1）相离——线段在圆的外部，彼此不相交，例如图中线段 1。

（2）相切——线段和圆相切，有一个切点，例如图中线段 2。

（3）包含——线段两个端点均在圆内部，线段和圆也彼此不相交，例如图中线段 3。

（4）相交——线段和圆有两个交点，例如图中线段 4。

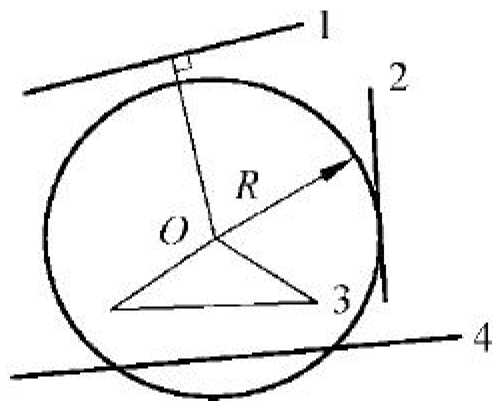


图 4.3-1　线段和圆的位置关系

为了实现圆窗口对线段的裁剪，需要判断线段和圆的位置关系。线段和圆的位置关系可以通过计算圆心到线段的垂直距离，并与圆的半径进行比较来确定。如果圆心到线段的距离大于圆的半径，则线段和圆处于相离情况；其距离等于半径时，则线段和圆处于相切情况。这两种情况下，线段都在圆窗口的外部，整条线段都被裁剪掉。当圆心到线段的距离小于圆的半径时，再进一步判断线段和圆的位置关系，如果线段两个端点到圆心的距离均小于圆的半径，端点都在圆的内部，则线段和圆是包含关系，线段在圆内，圆裁剪时，整个线段都保留。其余情况下线段和圆是相交关系，需要计算线段和圆的交点，当交点在线段两个端点之间，而不是在线段延长线上时，是有效交点。这时，交点将线段分成几部分，需判断哪一部分线段在圆内部，并保留圆内的线段部分。当线段的两个端点到圆心的距离都大于圆的半径时，两个端点都在圆的外部，两个交点之间的线段部分在圆的内部，则保留两个交点之间

的线段部分。如果线段与圆只有一个有效交点，则保留该交点与线段在圆内的端点之间的这一部分线段。

上述的圆窗口线段裁剪思路中，需要计算圆心到线段的距离以及线段与圆的交点。设线段 P_0P_1 的两个端点分别为 $P_0(x_0,y_0)$，$P_1(x_1,y_1)$，圆的半径是 R，圆心为 $P_c(x_c,y_c)$。线段的隐式方程为 $ax+by+c=0$（式中，$a=y_0-y_1$，$b=x_1-x_0$，$c=x_0y_1-x_1y_0$），圆心 $P_c(x_c,y_c)$ 到直线的垂直距离为

$$d=\left|\frac{ax_c+by_c+c}{\sqrt{a^2+b^2}}\right|$$

如果 $d\geqslant R$，则线段在裁剪窗口外，裁剪时，整个线段都被裁剪掉。

在计算线段和圆的交点时，将线段 P_0P_1 所在直线的参数方程

$$\begin{cases}x=(1-t)x_0+tx_1\\ y=(1-t)y_0+ty_1\end{cases}$$

代入圆的方程 $(x-x_c)^2+(y-y_c)^2=R^2$ 中，得

$$[(1-t)x_0+tx_1-x_c]^2+[(1-t)y_0+ty_1-y_c]^2=R^2$$

整理为

$$[(x_1-x_0)^2+(y_1-y_0)^2]t^2-2[(x_1-x_0)(x_c-x_0)+(y_1-y_0)(y_c-y_0)]t+$$
$$(x_c-x_0)^2+(y_c-y_0)^2-R^2=0$$

令

$$A=(x_1-x_0)^2+(y_1-y_0)^2,\quad B=2[(x_1-x_0)(x_c-x_0)+(y_1-y_0)(y_c-y_0)]$$
$$C=(x_c-x_0)^2+(y_c-y_0)^2-R^2$$

则上式简化为一元二次方程

$$At^2-Bt+C=0$$

求解该方程，得

$$t=\frac{B\pm\sqrt{B^2-4AC}}{2A}$$

根据交点的参数值 t，即可计算出线段和圆的交点。

实际上，单独利用上述的线段和圆联立的一元二次方程也可以判断圆与直线的位置，并实现圆形窗口对线段的裁剪。

令 $\Delta=B^2-4AC$。

当 $\Delta<0$ 时，方程无解，即直线和圆没有交点，线段和圆是相离的情况。

当 $\Delta=0$ 时，方程只有一个解，即直线和圆只有一个交点，线段和圆是相切的情况。

当 $\Delta>0$ 时，方程的两个解 t_1、t_2 的情况：

当 $t_1,t_2>1$ 或者 $t_1,t_2<0$ 时，说明两个交点都在线段的延长线上，那么线段和圆是包含的情况。

当 $0\leqslant t_1,t_2\leqslant 1$ 时，两个交点都在线段的两个端点之间，两个交点之间的线段部分在圆的内部。

当 t_1 和 t_2 中一个在[0,1]区间、一个不在[0,1]区间时，参数值 t 在[0,1]区间的交点和端点距离圆心最近的那部分线段在圆内部。

通过上述一元二次方程的求解方法，也可以实现圆窗口对线段的裁剪。由于平方根的计算比较耗时，在算法中应尽量减少求平方根的次数。上述在判断线段和圆是包含关系时，可以利用线段两个端点到圆心距离的平方与半径的平方进行比较，不必用参数值 t_1、t_2 判断。

4.3.3 任意多边形窗口对圆的裁剪

圆裁剪的第二种类型是任意形状的多边形作为裁剪窗口，对一个整圆或者一个圆弧段进行裁剪，将圆在多边形区域外边的部分剪掉，只保留多边形内部的圆弧部分。任意形状的多边形窗口对圆的裁剪方法也适用于矩形窗口的圆裁剪，因此具有一定的通用性。

多边形裁剪窗口和圆弧的位置关系也有整个圆弧在窗口外、整个圆弧在窗口内以及部分圆弧在窗口内等多种情况。当整个圆都在多边形裁剪窗口外部时，整个圆都被裁剪掉；当整个圆都在多边形裁剪窗口内部时，整个圆都保留；当圆和多边形窗口的边相交时，需要求交点，并判断多边形内部的圆弧部分，再进行裁剪。

多边形对圆的裁剪和多边形对多边形的裁剪非常类似，都是保留裁剪窗口内部的部分，将裁剪窗口外边的形状裁剪掉；不同之处是对多边形裁剪后，如果被裁剪多边形不再封闭，需要将裁剪窗口的部分边界加入被裁剪多边形中，使多边形裁剪后仍然是封闭的形状，而对圆裁剪后，成为圆弧段即可。

假想圆是一个由 0°～360°的圆弧构成的"多边形"，那么多边形裁剪窗口的圆裁剪就可看作是多边形对这个特殊"多边形"的裁剪，前述的多边形裁剪理论和算法即可应用于多边形窗口对圆的裁剪。在多边形裁剪方法中，Weiler-Atherton 裁剪算法是一种非常有效的任意形状多边形裁剪解决方案，利用 Weiler-Atherton 裁剪算法的思路也可以实现任意形状多边形窗口对圆的裁剪。

图 4.3-2 所示为多边形裁剪圆的情况，根据 Weiler-Atherton 裁剪算法的理论，多边形与圆的交点有进入多边形内部的进点和离开多边形内部的出点两种类型，将交点插入两个多边形的顶点序列中，按照多边形的方向，顺序利用进点和出点特性，在两个多边形顶点序列中交替记录在裁剪多边形内部的顶点，所得结果即为裁剪后的多边形顶点序列。在对圆裁剪时，裁剪后的圆弧段无须封闭，裁剪后的顶点序列中不用记录多边形的顶点，顶点均为圆上的交点，这样，只需将交点在圆方向上顺序排序，相邻的进点和出点之间的圆弧段即为裁剪多边形内部的圆弧段。在图 4.3-2 中，设置多边形的方向为顺时针，圆为逆时针，顺序计算多边形与圆的交点得 I_0、I_1、I_2、I_3、I_4、I_5 其中，I_0、I_2、I_4 为进点，I_1、I_3、I_5 为出点。将交点按圆的方向即逆时针方向排序，交点顺序为 I_2、I_1、I_0、I_5、I_4、I_3，依次获得相邻的进出点圆弧段 I_2-I_1、I_0-I_5、I_4-I_3，它们在多边形内部，裁剪后保留。

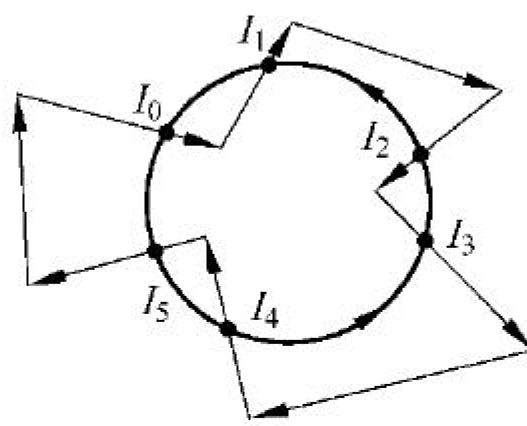


图 4.3-2 多边形裁剪圆

实现上述算法有两个关键点：一是计算多边形的边与圆的交点，并判断交点是进点还是出点；二是交点在圆上正确排序。

在计算多边形的边与圆的交点以及判断交点是进点还是出点时，首先要分析边与圆的

位置关系，根据 4.3.2 节“圆形窗口的线段裁剪”的相关内容，利用边所在线段的参数方程与圆方程联合的一元二次方程来判断交点情况。当有交点时，可以利用交点在线段上的参数值 t 的大小来判断该交点是进点还是出点。

设线段 P_0P_1 是多边形的一条边，两个端点分别为 $P_0(x_0, y_0)$，$P_1(x_1, y_1)$，边的方向是由 P_0 指向 P_1，该边所在直线的参数方程为

$$\begin{cases} x = (1-t)x_0 + tx_1 \\ y = (1-t)y_0 + ty_1 \end{cases}$$

当参数值 $t \in [0,1]$ 时，点在直线上的 P_0 和 P_1 之间，即在多边形的边上；当 $t<0$ 时，点在直线上靠近 P_0 的延长线上；当 $t>1$ 时，点在直线上靠近 P_1 的延长线上。那么，对于多边形的边来说，如已知边所在直线上一个点的参数值 t 的大小，则该点在边上的位置也可以确定下来。

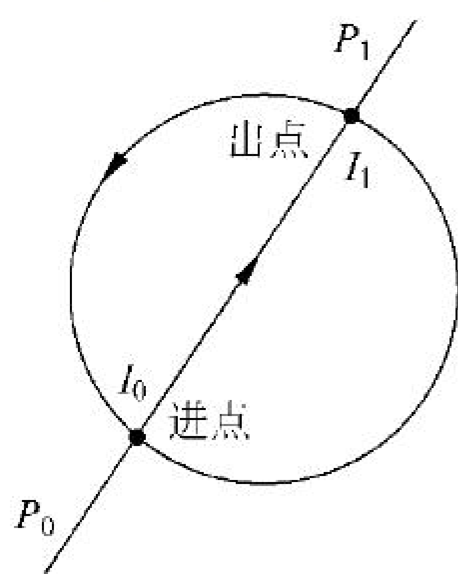


图 4.3-3　圆与有向线段的交点

设圆的半径是 R，圆心为 $P_c(x_c, y_c)$，逆时针方向。计算圆与上述多边形边 P_0P_1 所在直线的交点，如相交，记交点为 I_0 和 I_1，如图 4.3-3 所示。设两个交点在直线的参数方程中对应的参数值分别为 t_0 和 t_1，分析图 4.3-3 中的交点，直线的方向是由 P_0 指向 P_1，则沿着直线走向的右侧是所对应的多边形的内部方向，当圆是逆时针方向时，圆在交点 I_0 处进入多边形内部，I_0 为进点，在交点 I_1 处离开多边形，I_1 为出点。这时 I_0 和 I_1 对应的参数值 t_0 和 t_1 的大小关系是

$$t_0 < t_1$$

在直线和圆的方向都是上述设定的情况下，无论直线和圆的相对位置如何改变，这种进点和出点的参数值的大小关系均不会发生改变。那么，通过比较两个交点在直线上的参数值大小，就可以判断哪个交点是进点或出点。

需要注意的是，这里的交点是指多边形边所在直线与圆的交点，并不一定是多边形边与圆的交点，交点可能在多边形边上也可能在边延长线上。当交点在边上时，称为实交点；当交点在边延长线上时，称为虚交点。当多边形的边与圆没有实交点时，不用计算参数值。只有在有实交点的情况下，才计算两个交点的参数值，并比较大小，判断实交点是进点还是出点。判断多边形边和圆是否有实交点可参考 4.3.2 节“圆形窗口的线段裁剪”的相关内容。

计算出多边形与圆的交点后，还需要在圆周方向上对交点进行正确排序，从而依次获得进出点圆弧段。交点在圆周方向排序时，不用通过计算和比较每个交点到圆心的圆弧角排序，当交点坐标值 $y \geqslant 0$ 时，按每个交点到圆上点 $(R,0)$ 的距离（或者距离的平方）排序；当交点坐标值 $y<0$ 时，按照距离的反向排序，或者按交点到圆上点 $(-R,0)$ 的距离（或者距离的平方）排序，如图 4.3-4 所示。

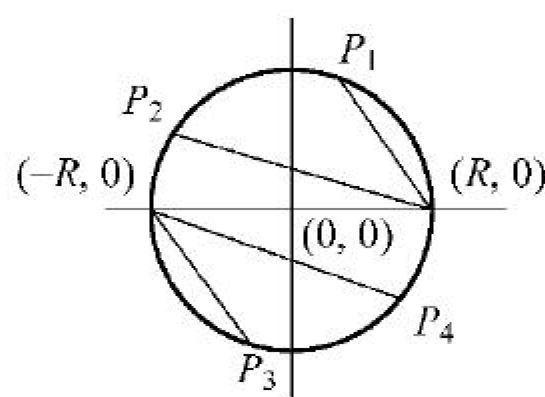


图 4.3-4　圆周方向排序

上述任意多边形作为窗口对圆进行裁剪的代码如下：

```
/************************************************************************
ArcClipping: 任意多边形对圆的裁剪函数
cr: 圆; m_CutPolyLine_array: 任意裁剪多边形; m_arc_array1: 裁剪后的圆弧段集合
************************************************************************/
```

```
void ArcClipping(CCircle& cr, CArray < CLine, CLine > &m_CutPolyLine_array, CArray < CCircle,
CCircle > &m_arc_array1) {
    CArray < CInterPoint, CInterPoint > m_circle_point_Array, m_cut_point_Array;  //加入交点
//后的圆弧交点序列和排序后裁剪多边形序列
    //1.设置裁剪多边形方向(顺时针)
    SortForPolyline(m_CutPolyLine_array, 0, m_cut_point_Array);
    //2.逐边和圆求交点,交点设置为进点或者出点,并插入圆弧交点序列中
    CInterPoint ptA, ptB, pt_0, pt_1, interPt;
    CArray < CInterPoint, CInterPoint > m_InterPt_Array;  //记录当前边相交点的集合
    int pFlag = 1;  //记录圆和多边形的位置关系,0: 有交点,1: 内部,2: 外部,默认在内部
    int pFlagtmp = 1;
    for(int j = 0; j < m_cut_point_Array.GetSize() - 1; j++){
        ptA = m_cut_point_Array.GetAt(j);
        ptB = m_cut_point_Array.GetAt(j + 1);
        //判断线段 ptA - ptB 和圆是否有交点,如有交点,有几个,按 t 大小排序
        pFlagtmp = GetInterPt(ptA, ptB, cr, m_InterPt_Array);  //计算交点
        if(pFlagtmp == 0){
            while(m_InterPt_Array.GetSize()> 0){
                pt_0 = m_InterPt_Array.GetAt(0);
                //圆周上点逆时针排序
                OrderToCirclePt(m_circle_point_Array, pt_0, cr.Opt, cr.rLength);
                m_InterPt_Array.RemoveAt(0);
            }
            pFlag = pFlagtmp;
        }
    }
    if(pFlag == 0){                               //有交点,顺序取出进点和出点,构成圆弧段
        CCircle cr_New;
        cr_New.Opt = cr.Opt;
        cr_New.rLength = cr.rLength;
        cr_New.Type = 1;                          //圆弧
        int i = 0;
        int pt_flag = 0;
        int stopFlag = 0;                         //是否停止的标识符
        while(1){
            if(i < m_circle_point_Array.GetSize()){
                pt_1 = m_circle_point_Array.GetAt(i);
                if(pt_1.flag == 0)                //非进点/出点
                    i++;
                else  {
                    if(pt_1.flag == 1){           //进点
                        cr_New.Ept.x = pt_1.x;
                        cr_New.Ept.y = pt_1.y;
                        pt_flag = 1;
                        i++;
                        continue;                 //查找对应的出点
                    }
                    else if(pt_1.flag ==- 1){     //出点
                        if(pt_flag == 1){         //进点已经找到,则输出
                            cr_New.Spt.x = pt_1.x;
                            cr_New.Spt.y = pt_1.y;
```

```
                                m_arc_array1.Add(cr_New);  //输出
                                //开始查询下一个弧段
                                if(stopFlag == 1)
                                    break;             //从头又开始搜索,找到了出点后,则停止
                                else{
                                    pt_flag = 0;
                                    i++;
                                    continue;
                                }
                            }
                            else{
                                i++;
                                continue;
                            }
                        }
                    }
                }
                else {
                    if(pt_flag == 1){                  //有进点,需要找出点,从头再开始查找
                        i = 0;
                        stopFlag = 1;                  //需要找出点
                        continue;
                    }
                    else
                        break;                         //不再有进点,则停止
                }
            }
        }
        else if(pFlag == 1){
            m_arc_array1.Add(cr);                      //圆在多边形内部,保留,输出
        }
        else                                           //圆在多边形外部直接返回,表示裁剪掉
            return ;
    }
```

代码中的交点类结构 CInterPoint 和多边形方向排序函数 SortForPolyline()在 4.2.3 节“任意形状多边形的裁剪”中已经列出,本处不再重复。判断多边形边和圆的位置关系以及交点个数和交点是进点还是出点的函数代码如下:

```
/* ptA: 线段首点; ptB: 线段末点; cr: 圆; m_InterPt_Array: 交点集合; 返回值: 1 表示圆在该线段的内部,2 表示圆在该线段的外部,0 表示圆和线段有交点 */
int GetInterPt ( CInterPoint &ptA, CInterPoint &ptB, CCircle& cr, CArray < CInterPoint, CInterPoint > &m_InterPt_Array){
    m_InterPt_Array.RemoveAll();                    //清空当前裁剪边的交点序列
    double A = (double)(ptB.x - ptA.x) * (ptB.x - ptA.x) + (ptB.y - ptA.y) * (ptB.y - ptA.y);
    double B = 2 * ((double)(ptB.x - ptA.x) * (cr.Opt.x - ptA.x) + (ptB.y - ptA.y) * (cr.Opt.y - ptA.y));
    double C = (double)(cr.Opt.x - ptA.x) * (cr.Opt.x - ptA.x) + (cr.Opt.y - ptA.y) * (cr.Opt.y - ptA.y) - cr.rLength * cr.rLength;
    double Delta = B * B - 4 * A * C;
```

```
    //1.判断是否没有交点
    if(Delta<=0)                              //直线与圆相离或者相切,没有交点
        return 1;                             //返回 1 表示圆在该线段的内部
    //2.判断是否圆包含线段,判断两个端点到圆心的距离和半径大小
double AO=(double)(ptA.x-cr.Opt.x)*(ptA.x-cr.Opt.x)+(ptA.y-cr.Opt.y)*(ptA.y-cr.Opt.y);
    double BO=(ptB.x-cr.Opt.x)*(ptB.x-cr.Opt.x)+(ptB.y-cr.Opt.y)*(ptB.y-cr.Opt.y);
    if(AO<=cr.rLength*cr.rLength&&BO<cr.rLength*cr.rLength)
        return 2;                             //返回 2 表示圆在该线段的外部
    //有交点,则计算 t 值
    double t1,t2;
    double DltaRoot=sqrt(Delta);
    t1=(B-DltaRoot)/(2*A);
    t2=(B+DltaRoot)/(2*A);
    //因为 t1<t2,所以 t1 是进点 1,t2 是出点 -1
    CInterPoint pt0;
    if(t1>=0&&t1<=1)  {                       //进点
        pt0.x=(int)((1-t1)*ptA.x+t1*ptB.x+0.5);
        pt0.y=(int)((1-t1)*ptA.y+t1*ptB.y+0.5);
        pt0.flag=1;                           //进点
        m_InterPt_Array.Add(pt0);
    }
    if(t2>=0&&t2<=1)  {                       //出点
        pt0.x=(int)((1-t2)*ptA.x+t2*ptB.x+0.5);
        pt0.y=(int)((1-t2)*ptA.y+t2*ptB.y+0.5);
        pt0.flag=-1;                          //出点
        m_InterPt_Array.Add(pt0);
    }
    return 0;                                 //返回 0 表示圆和线段相交
}
```

在圆周上交点按逆时针方向的排序函数代码如下：

```
/*m_Pt_Array: 交点集合; m_Pt: 带排序的交点; cPt: 圆心; R: 半径*/
void OrderToCirclePt(CArray<CInterPoint,CInterPoint>&m_Pt_Array, CInterPoint& m_Pt,
CPoint cPt,int R){
    int x,y,xi,yi;
    double L,Li;
    x=m_Pt.x-cPt.x;
    y=m_Pt.y-cPt.y;
    if(y>0)
        L=(x-R)*(x-R)+y*y;
    else
        L=(x+R)*(x+R)+y*y;
    for(int i=0;i<m_Pt_Array.GetSize();i++){
        xi=m_Pt_Array.GetAt(i).x-cPt.x;
        yi=m_Pt_Array.GetAt(i).y-cPt.y;
        if(y*yi>=0){
            if(yi>0)
                Li=(xi-R)*(xi-R)+yi*yi;
```

```
                else
                    Li = (xi + R) * (xi + R) + yi * yi;
                if(L < Li){                              //插入,返回
                    m_Pt_Array.InsertAt(i,m_Pt);
                    return;  }
            }
            else  {
                if(y > 0){                               //插入,返回
                    m_Pt_Array.InsertAt(i,m_Pt);
                    return;
                }
            }
        }
        m_Pt_Array.Add(m_Pt);                            //交点值大,则加入尾部
    }
```

如图 4.3-5 所示为一个任意多边形和一个裁剪圆以及利用上述代码对多边形进行裁剪后的效果。

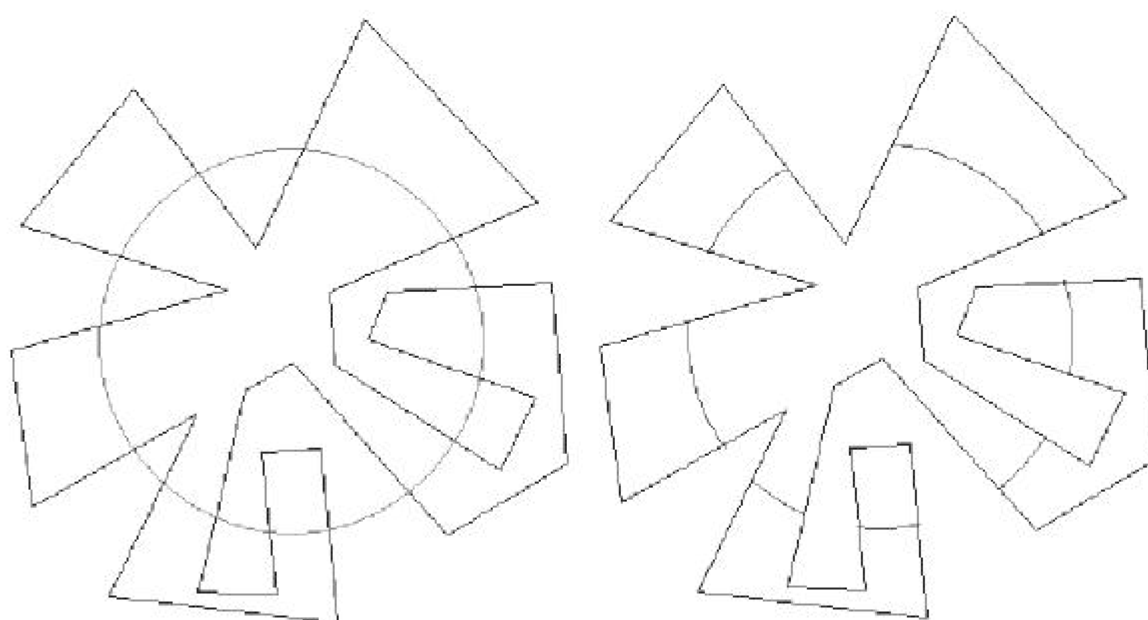


图 4.3-5　任意多边形窗口的圆裁剪

4.4　字符裁剪

字符也是图形的一种,它在输出过程中同样需要裁剪,当字符和文本部分在窗口内、部分在窗口外时,就需要将窗口外的部分裁剪掉。根据字符的生成及存储方式和具体应用要求,字符的裁剪分为三种方式。

1. 基于字符串精度裁剪

采用字符串方式裁剪时,将包围字符串的外接矩形对窗口做裁剪,当字符串的外接矩形整个在窗口内时,予以显示,否则不显示,如图 4.4-1(a)、(b)所示。

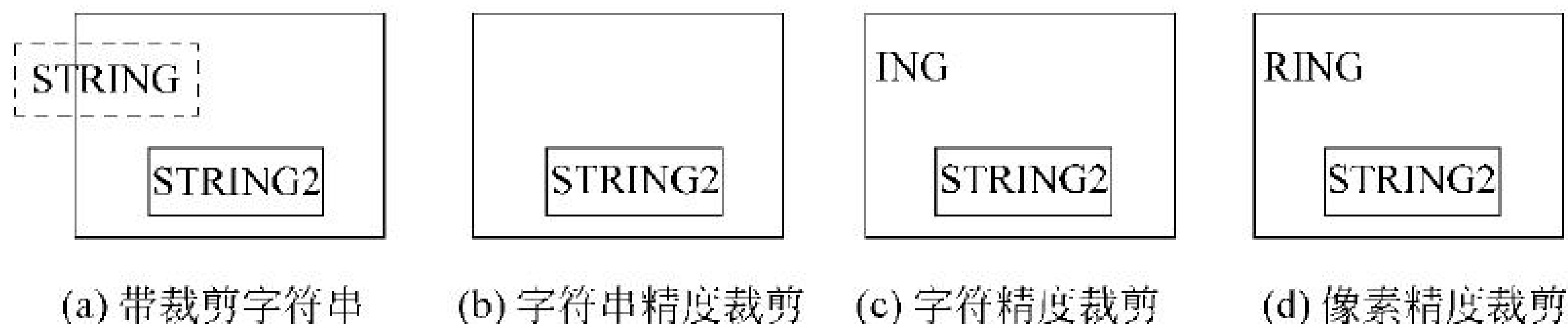


图 4.4-1　字符裁剪

2. 基于字符精度裁剪

采用字符方式裁剪时，将包围字符的外接矩形对窗口做裁剪，如某个字符的外接矩形整个落在窗口内，予以显示，否则不显示，如图 4.4-1(c)所示。

3. 基于笔画/像素精度的裁剪

对于点阵字符，构成字符的最小元素为像素，此时字符的裁剪转化为点裁剪。对于矢量字符，构成字符的最小元素是直线段(笔画)，这样字符的裁剪就转化为直线的裁剪，如图 4.4-1(d)所示。

第 5 章

图形变换

对于计算机生成的图形，有时为了获得更好的观察效果，需要将其旋转不同的角度、方向或者移动到指定的位置再在屏幕上显示，这些操作都称为图形变换。

图形变换是计算机图形学的数学基础之一，是计算机图形学中一个重要概念和理论，也是 CAD 系统必不可少的核心功能。在面向各种 CAD 的应用中，设计者从构图到观察和分析所设计的结果常常需要各种图形变换，图形变换也是实现动画仿真、构造虚拟现实的基础。图形变换有对二维图形的几何变换和对三维形体的几何变换两种类型，由于显示屏幕只能显示二维图形，因此对三维形体需要投影到显示屏幕上才能显示，即投影变换。各种图形处理过程都是通过相应的几何变换或投影变换来实现的。通过图形变换，可由简单图形生成复杂图形，可用二维图形表示三维形体，甚至可对静态图形经过快速变换而获得图形的动态显示效果。在实现图形变换时，不可避免地会使用交互技术实现图形拾取、对象捕捉及常用的鼠标操作等，所以，本章对图形交互技术也进行了一定的研究和算法实现。由于三维变换主要是针对三维形体的操作，所以，本章对三维形体的线框造型方法做了部分论述。在实现各种图形变换的过程中，矩阵运算是最基本的数学基础，本章首先对相关数学知识作一简单回顾。

5.1 图形变换的数学基础

5.1.1 矢量的定义及运算

矢量是一有向线段，具有方向和大小两个参数，设有两个矢量 $\boldsymbol{V}_1(x_1,y_1,z_1)$，$\boldsymbol{V}_2(x_2,y_2,z_2)$。相关矢量运算公式如下。

(1) 矢量的长度：

$$|\boldsymbol{V}_1| = \sqrt{(x_1^2 + y_1^2 + z_1^2)}$$

(2) 数乘矢量：

$$a\boldsymbol{V}_1 = (ax_1, ay_1, az_1)$$

(3) 两个矢量之和，方向如图 5.1-1 所示：

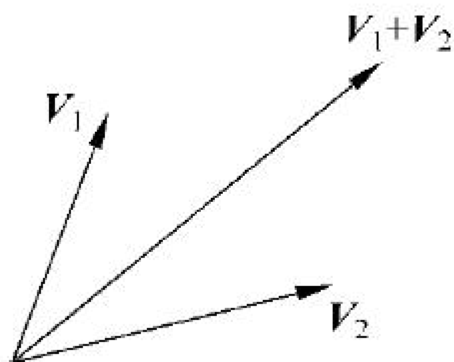


图 5.1-1 矢量之和

$$\boldsymbol{V}_1 + \boldsymbol{V}_2 = (x_1,y_1,z_1) + (x_2,y_2,z_2) = (x_1+x_2, y_1+y_2, z_1+z_2)$$

(4) 两个矢量的点积：

$$\boldsymbol{V}_1 \cdot \boldsymbol{V}_2 = |\boldsymbol{V}_1| \cdot |\boldsymbol{V}_2| \cos\theta = x_1x_2 + y_1y_2 + z_1z_2$$

其中，θ 为两个矢量之间的夹角。

(5) 两个矢量的叉积：

$$\boldsymbol{V}_1\times\boldsymbol{V}_2=\begin{bmatrix}\boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k}\\ x_1 & y_1 & z_1\\ x_2 & y_2 & z_2\end{bmatrix}$$

$$=(y_1z_2-z_1y_2,z_1x_2-x_1z_2,x_1y_2-y_1x_2)$$

两个矢量叉积的方向如图 5.1-2 所示。

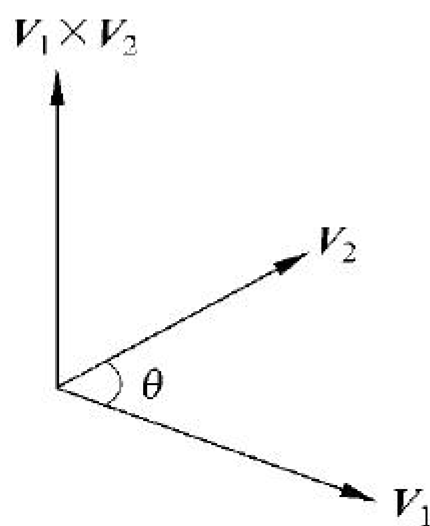


图 5.1-2 两矢量叉积

5.1.2 矩阵的定义及运算

设有一个 m 行 n 列矩阵 $\boldsymbol{A}$：

$$\boldsymbol{A}_{m\times n}=\begin{bmatrix}a_{11} & a_{12} & \cdots & a_{1n}\\ a_{21} & a_{22} & \cdots & a_{2n}\\ \vdots & \vdots & & \vdots\\ a_{m1} & a_{m2} & \cdots & a_{mn}\end{bmatrix}$$

其中，$(a_{i1}\quad a_{i2}\quad\cdots\quad a_{in})$为第 i 个行向量，$(a_{1j}\quad a_{2j}\quad\cdots\quad a_{mj})^{\mathrm{T}}$ 为第 j 个列向量，a_{ij} 为第 i 行第 j 列元素。相关矩阵运算公式如下。

(1) 矩阵的加法

设两个矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 都是 $m\times n$ 的，把它们对应位置的元素相加而得到的矩阵叫作 $\boldsymbol{A}$、$\boldsymbol{B}$ 的和，记为 $\boldsymbol{A}+\boldsymbol{B}$。只有两个矩阵的行数和列数都相同时才能相加。

$$\boldsymbol{A}+\boldsymbol{B}=\begin{bmatrix}a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n}\\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n}\\ \vdots & \vdots & & \vdots\\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn}\end{bmatrix}$$

(2) 数乘矩阵

用数 k 乘矩阵 $\boldsymbol{A}$ 的每一个元素而得的矩阵叫作 k 与 $\boldsymbol{A}$ 之积，记为 $k\boldsymbol{A}$：

$$k\boldsymbol{A}=\begin{bmatrix}ka_{11} & ka_{12} & \cdots & ka_{1n}\\ ka_{21} & ka_{22} & \cdots & ka_{2n}\\ \vdots & \vdots & & \vdots\\ ka_{m1} & ka_{m2} & \cdots & ka_{mn}\end{bmatrix}$$

(3) 矩阵的乘法

只有当前一矩阵的列数等于后一矩阵的行数时两个矩阵才能相乘，即

$$\boldsymbol{C}_{m\times n}=\boldsymbol{A}_{m\times p}\boldsymbol{B}_{p\times n}$$

矩阵 $\boldsymbol{C}$ 中的每一个元素 $C_{ij}=\sum_{k=1}^{p}a_{ik}b_{kj}$。

例如，设 $\boldsymbol{A}$ 为 2×3 的矩阵，$\boldsymbol{B}$ 为 3×2 的矩阵，则两者的乘积为

$$\boldsymbol{C}=\boldsymbol{AB}=\begin{bmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\end{bmatrix}\begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\\ b_{31} & b_{32}\end{bmatrix}$$

$$=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} & a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32}\\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}\end{bmatrix}$$

(4) 单位矩阵

一个 $n\times n$ 的矩阵，如果它的对角线上的各个元素均为 1，其余元素都为 0，则该矩阵称为单位矩阵，记为 $\boldsymbol{I}_n$。对于任意 $m\times n$ 的矩阵恒有

$$\boldsymbol{A}_{m\times n}\boldsymbol{I}_n=\boldsymbol{A}_{m\times n},\quad \boldsymbol{I}_m\boldsymbol{A}_{m\times n}=\boldsymbol{A}_{m\times n}$$

(5) 矩阵的转置

交换一个矩阵 $\boldsymbol{A}_{m\times n}$所有的行列元素，那么所得到的 $n\times m$ 矩阵被称为原有矩阵的转置，记为 $\boldsymbol{A}^{\mathrm{T}}$：

$$\boldsymbol{A}^{\mathrm{T}}=\begin{bmatrix}a_{11} & a_{21} & \cdots & a_{m1}\\ a_{12} & a_{22} & \cdots & a_{m2}\\ \vdots & \vdots & & \vdots\\ a_{1n} & a_{2n} & \cdots & a_{mn}\end{bmatrix}$$

(6) 矩阵的逆

对于一个 $n\times n$ 的方阵 $\boldsymbol{A}$，如果存在一个 $n\times n$ 的方阵 $\boldsymbol{B}$，使得 $\boldsymbol{AB}=\boldsymbol{BA}=\boldsymbol{I}_n$，则称 $\boldsymbol{B}$ 是 $\boldsymbol{A}$ 的逆，记为 $\boldsymbol{B}=\boldsymbol{A}^{-1}$，$\boldsymbol{A}$ 则被称为非奇异矩阵。矩阵的逆是相互的，$\boldsymbol{A}$ 同样也可记为 $\boldsymbol{A}=\boldsymbol{B}^{-1}$，$\boldsymbol{B}$ 也是一个非奇异矩阵。任何一个非奇异矩阵有且只有一个逆矩阵。

(7) 矩阵运算的基本性质

① 矩阵的加法适合交换律与结合律：$\boldsymbol{A}+\boldsymbol{B}=\boldsymbol{B}+\boldsymbol{A}$，$\boldsymbol{A}+(\boldsymbol{B}+\boldsymbol{C})=(\boldsymbol{A}+\boldsymbol{B})+\boldsymbol{C}$；

② 数乘矩阵适合分配律与结合律：$a(\boldsymbol{A}+\boldsymbol{B})=a\boldsymbol{A}+a\boldsymbol{B}$，$a(\boldsymbol{AB})=(a\boldsymbol{A})\boldsymbol{B}=\boldsymbol{A}(a\boldsymbol{B})$；

③ 矩阵的乘法适合结合律：$\boldsymbol{A}(\boldsymbol{BC})=(\boldsymbol{AB})\boldsymbol{C}$；

④ 矩阵的乘法对加法运算适合分配律：$(\boldsymbol{A}+\boldsymbol{B})\boldsymbol{C}=\boldsymbol{AC}+\boldsymbol{BC}$，$\boldsymbol{C}(\boldsymbol{A}+\boldsymbol{B})=\boldsymbol{CA}+\boldsymbol{CB}$；

⑤ 矩阵的乘法不适合交换率：$\boldsymbol{AB}\neq\boldsymbol{BA}$。

5.1.3 齐次坐标

用 $n+1$ 维向量表示 n 维空间点的方法称为齐次坐标表示法，而此 $n+1$ 维向量称为此 n 维空间点的齐次坐标。

用三维向量 $[x\quad y\quad 1]$ 表示二维平面点 (x,y)，$[x\quad y\quad 1]$ 称为点 $P(x,y)$ 的齐次坐标。

二维点 (x,y) 的齐次坐标一般式可写成

$$[Hx\quad Hy\quad H],\quad H\neq 0$$

$[x\quad y\quad 1]$ 称为点 $P(x,y)$ 的规格化齐次坐标。用点的齐次坐标一般式求解点的规格化齐次坐标的过程如下：

$$[Hx \quad Hy \quad H] \Rightarrow \left[\frac{Hx}{H} \quad \frac{Hy}{H} \quad \frac{H}{H}\right] = [x \quad y \quad 1]$$

这个过程称为齐次坐标的正常化。

齐次坐标表示法提供了用矩阵运算把二维、三维甚至高维空间中的一个点集从一个坐标系变换到另一个坐标系的有效方法，而且，采用齐次坐标，可以实现图形变换矩阵形式的统一。

5.2 二维图形几何变换

5.2.1 二维几何变换概述

因为各种图形都可看成点的集合，因此图形变换可以归结为图形上点的坐标变换，而点在变换前后的坐标关系一般用几何的方法即可求得。

如图5.2-1所示，$P(x,y)$是平面坐标系xOy中的点，P点到坐标原点的距离为ρ，线段OP和x轴的夹角为α。则P点的坐标(x,y)为

$$\begin{cases} x = \rho\cos\alpha \\ y = \rho\sin\alpha \end{cases}$$

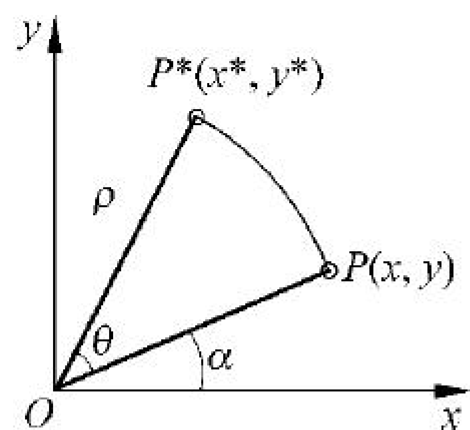


图5.2-1 点的坐标变换

P点绕原点旋转角度θ后到P^*点，则P^*点的坐标(x^*,y^*)为

$$\begin{cases} x^* = \rho\cos(\theta+\alpha) = \rho\cos\alpha\cos\theta - \rho\sin\alpha\sin\theta = x\cos\theta - y\sin\theta \\ y^* = \rho\sin(\theta+\alpha) = \rho\sin\alpha\cos\theta + \rho\cos\alpha\sin\theta = x\sin\theta + y\cos\theta \end{cases}$$

因此，旋转后点的坐标与旋转前点的坐标关系为

$$\begin{cases} x^* = x\cos\theta - y\sin\theta \\ y^* = x\sin\theta + y\cos\theta \end{cases}$$

也可以写成矩阵形式：

$$[x^* \quad y^*] = [x \quad y]\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

对一个平面图形的旋转变换，即对n个点的点集进行变换，其矩阵形式为

$$\begin{bmatrix} x_1^* & y_1^* \\ x_2^* & y_2^* \\ \vdots & \vdots \\ x_n^* & y_n^* \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

将上式简写为

$$\boldsymbol{P}^* = \boldsymbol{PT}$$

其中，$\boldsymbol{P}$为变换前各点的坐标矩阵，$\boldsymbol{P}^*$为变换后各点的坐标矩阵，$\boldsymbol{T}$称为旋转变换矩阵。将$\boldsymbol{T}$写成一般的形式：

$$\boldsymbol{T}=\begin{bmatrix}a & b\\ c & d\end{bmatrix}$$

则

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}a & b\\ c & d\end{bmatrix}=[ax+cy \quad bx+dy]$$

即

$$\begin{cases}x^*=ax+cy\\ y^*=bx+dy\end{cases}$$

这时，变换矩阵 $\boldsymbol{T}$ 不仅适用于点的旋转变换，改变矩阵 $\boldsymbol{T}$ 中各元素的值，也可以实现其他变换。对于各种变换，关键是确定变换矩阵 $\boldsymbol{T}$ 中各元素的值。

5.2.2 二维图形基本变换

基本变换是指相对坐标原点、坐标轴或坐标面的变换，包括平面图形的比例、镜像、错移、旋转、平移等变换。

1. 恒等变换

在变换矩阵 $\boldsymbol{T}=\begin{bmatrix}a & b\\ c & d\end{bmatrix}$中，令 $a=d=1,b=c=0$，即 $\boldsymbol{T}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$，变换矩阵为一单位阵，用此变换矩阵对点进行变换：

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=[x \quad y],\quad 即 \quad \begin{cases}x^*=x\\ y^*=y\end{cases}$$

点在变换前后的坐标没有改变，即点的位置没有变化，这种变换称为恒等变换，其单位矩阵又叫作恒等变换矩阵。

2. 比例变换

在变换矩阵 $\boldsymbol{T}=\begin{bmatrix}a & b\\ c & d\end{bmatrix}$中，令 $b=c=0$，即 $\boldsymbol{T}=\begin{bmatrix}a & 0\\ 0 & d\end{bmatrix}$，用此变换矩阵对点进行变换：

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}a & 0\\ 0 & d\end{bmatrix}=[ax \quad dy],\quad 即 \quad \begin{cases}x^*=ax\\ y^*=dy\end{cases}$$

点的坐标在 x 方向放大为原先的 a 倍，在 y 方向放大为原先的 d 倍，a、d 分别称为 x、y 向的比例因子，这种变换称为比例变换。

根据 a、d 的取值不同，比例变换有以下几种形式：

(1) 若 $a=d=1$，则变换矩阵 $\boldsymbol{T}$ 为一单位阵，称为恒等变换；

(2) 若 $a=d>1$，则图形沿 x、y 方向等比例放大；

(3) 若 $a=d<1$，则图形沿 x、y 方向等比例缩小；

(4) 若 $a\neq d$，则图形将产生变形——畸变；

(5) 若 $a=1$，图形只沿 y 向放大或缩小；

(6) 若 $d=1$，图形只沿 x 向放大或缩小；

(7) 若 $a=0$,图形压缩到 y 轴上;

(8) 若 $d=0$,图形压缩到 x 轴上。

3. 镜像变换(对称或反射变换)

1) 对 x 轴的镜像变换

在变换矩阵 $\boldsymbol{T}$ 中,取 $a=1,d=-1,b=c=0$,即变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$,对点进行变换:

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}=[x \quad -y], \quad 即 \quad \begin{cases}x^*=x\\y^*=-y\end{cases}$$

此变换为对 x 轴的镜像变换,如图 5.2-2 所示。

2) 对 y 轴的镜像变换

对 y 轴的镜像变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}-1 & 0\\0 & 1\end{bmatrix}$,则

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}-1 & 0\\0 & 1\end{bmatrix}=[-x \quad y], \quad 即 \quad \begin{cases}x^*=-x\\y^*=y\end{cases}$$

对 y 轴的镜像变换如图 5.2-3 所示。

3) 对坐标原点的镜像变换

对坐标原点的镜像变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}-1 & 0\\0 & -1\end{bmatrix}$,则

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}-1 & 0\\0 & -1\end{bmatrix}=[-x \quad -y], \quad 即 \quad \begin{cases}x^*=-x\\y^*=-y\end{cases}$$

对坐标原点的镜像变换如图 5.2-4 所示。

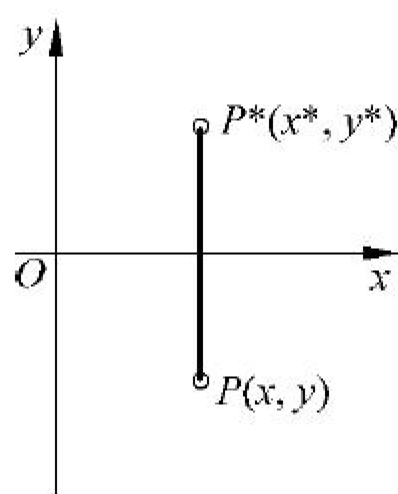


图 5.2-2　x 轴的镜像变换

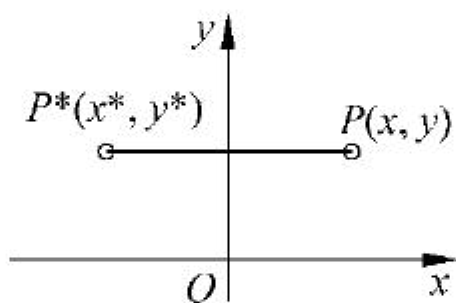


图 5.2-3　y 轴的镜像变换

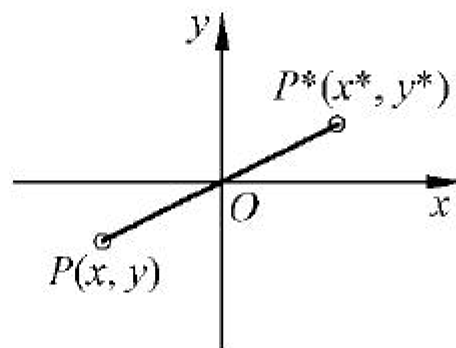


图 5.2-4　原点的镜像变换

4) 对 45°线(即 $y=x$ 轴)的镜像变换

镜像变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$,即$\begin{cases}x^*=y\\y^*=x\end{cases}$,如图 5.2-5 所示。

5) 对−45°线(即 $y=-x$ 轴)的镜像变换

镜像变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}0 & -1\\-1 & 0\end{bmatrix}$,即$\begin{cases}x^*=-y\\y^*=-x\end{cases}$,如图 5.2-6 所示。

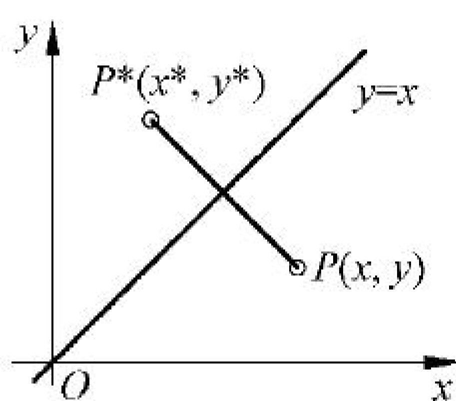


图 5.2-5 对 45°线的镜像变换

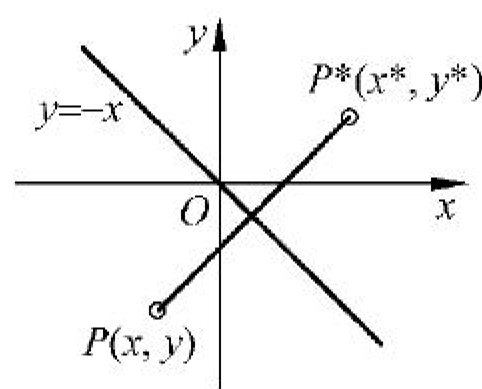


图 5.2-6 对－45°线的镜像变换

6）错移变换

沿 x 轴错移——在变换矩阵 $\boldsymbol{T}$ 中，令 $a=d=1, b=0$，即变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}1 & 0\\ c & 1\end{bmatrix}$，则

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}1 & 0\\ c & 1\end{bmatrix}=[x+cy \quad y], \quad 即 \quad \begin{cases}x^*=x+cy\\ y^*=y\end{cases}$$

变换后点的 y 坐标不变，相当于点沿 x 轴平移了 cy 距离(图 5.2-7)。点的平移量依赖于点的 y 坐标，而且与变换前点的 y 坐标成线性比例关系。

沿 y 轴错移——在变换矩阵 $\boldsymbol{T}$ 中，令 $a=d=1, c=0$，即变换矩阵为 $\boldsymbol{T}=\begin{bmatrix}1 & b\\ 0 & 1\end{bmatrix}$，则

$$[x^* \quad y^*]=[x \quad y]\begin{bmatrix}1 & b\\ 0 & 1\end{bmatrix}=[x \quad y+bx], \quad 即 \quad \begin{cases}x^*=x\\ y^*=y+bx\end{cases}$$

变换后点的 x 坐标不变，相当于点沿 y 轴平移了 bx 距离(图 5.2-8)。点的平移量依赖于点的 x 坐标，与变换前点的 x 坐标成线性比例关系。

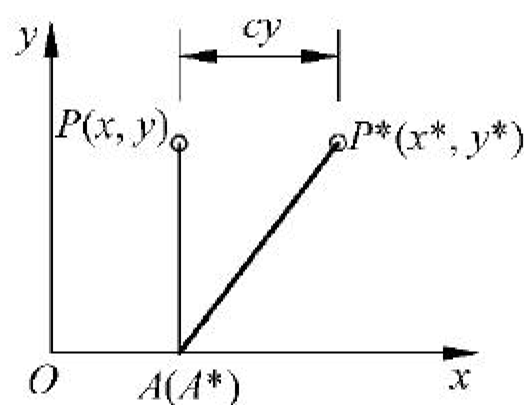


图 5.2-7 沿 x 轴错移

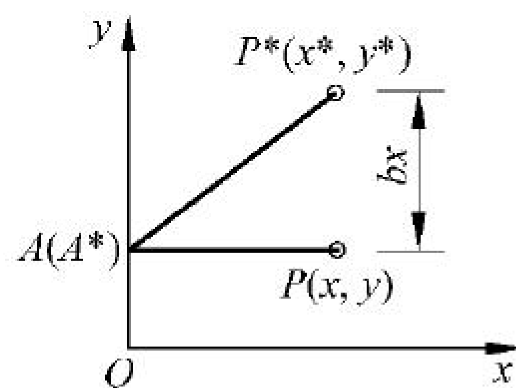


图 5.2-8 沿 y 轴错移

例 5.1 正方形 $ABCD$ 四个顶点坐标分别为 $A(0,0)$、$B(16,0)$、$C(16,16)$、$D(0,16)$，对此正方形进行沿 x 轴的错移变换，取 $c=0.5$，则

$$\begin{matrix}A\\B\\C\\D\end{matrix}\begin{bmatrix}0 & 0\\ 16 & 0\\ 16 & 16\\ 0 & 16\end{bmatrix}\begin{bmatrix}1 & 0\\ 0.5 & 1\end{bmatrix}=\begin{bmatrix}0 & 0\\ 16 & 0\\ 24 & 16\\ 8 & 16\end{bmatrix}\begin{matrix}A^*\\B^*\\C^*\\D^*\end{matrix}$$

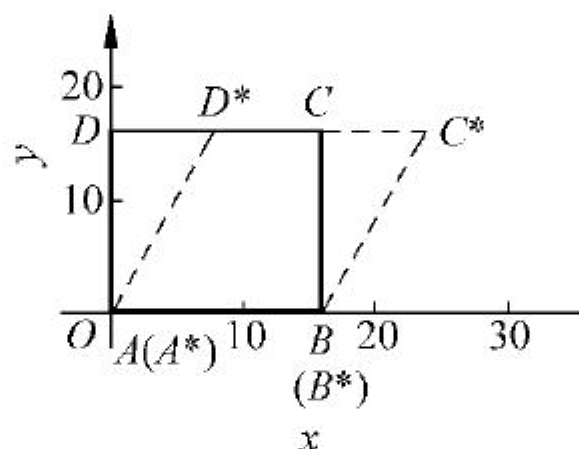


图 5.2-9 错移变换

错移变换前后的效果如图 5.2-9 所示。

7）旋转变换

由本节前述可知，旋转变换的变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}$$

其中，θ 为旋转角。

例 5.2 长方形 $ABCD$ 四个顶点的坐标分别为 $A(0,0)$、$B(20,0)$、$C(20,15)$、$D(0,15)$，绕坐标原点逆时针旋转 30°，则

$$\begin{matrix}A\\B\\C\\D\end{matrix}\begin{bmatrix}0 & 0\\20 & 0\\20 & 15\\0 & 15\end{bmatrix}\begin{bmatrix}\cos30^\circ & \sin30^\circ\\-\sin30^\circ & \sin30^\circ\end{bmatrix}=\begin{bmatrix}0 & 0\\17.32 & 1\\9.82 & 22.99\\-7.5 & 12.99\end{bmatrix}\begin{matrix}A^*\\B^*\\C^*\\D^*\end{matrix}$$

旋转变换效果如图 5.2-10 所示。

8）平移变换

平移变换用齐次坐标表示。对点 $P(x,y)$ 作平移变换，其 x 向平移量为 l，y 向平移量为 m，变换后的点为 $P^*(x^*,y^*)$，则平移变换的解析式为 $\begin{cases}x^*=x+l\\y^*=y+m\end{cases}$，这时，用 $\boldsymbol{T}=\begin{bmatrix}a & b\\c & d\end{bmatrix}$ 将不能表示平移变换，可以用齐次坐标形式表示。变换矩阵形式为

$$[x^* \quad y^* \quad 1]=[x \quad y \quad 1]\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\l & m & 1\end{bmatrix}=[x+l \quad y+m \quad 1]$$

平移变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\l & m & 1\end{bmatrix}$$

平移变换效果如图 5.2-11 所示。

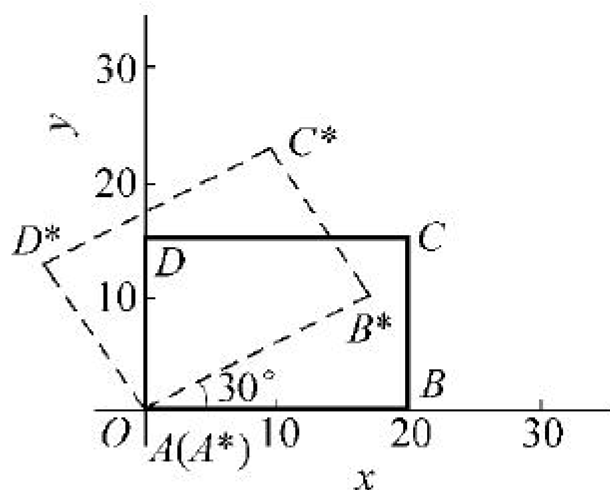


图 5.2-10 旋转变换

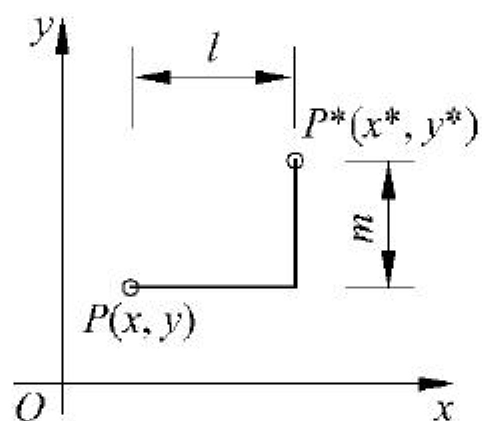


图 5.2-11 平移变换

9）二维基本变换的统一形式

采用齐次坐标，可以将变换矩阵形式统一。将上述所有变换都采用齐次坐标形式，则二维图形基本变换的统一形式为

$$[x^* \quad y^* \quad 1]=[x \quad y \quad 1]\begin{bmatrix}a & b & p\\c & d & q\\l & m & s\end{bmatrix}$$

变换矩阵为

$$T=\begin{bmatrix} a & b & p \\ c & d & q \\ l & m & s \end{bmatrix}$$

变换矩阵 $\boldsymbol{T}$ 中左上角的 2×2 子矩阵，即 a、b、c、d，可使图形产生比例、镜像、错移、旋转等基本变换；

变换矩阵 $\boldsymbol{T}$ 中左下角的 1×2 子矩阵，即 l、m，可使图形产生平移变换，其中 l、m 分别是 x 向和 y 向的平移量，在作其他变化时需使 $l=0$，$m=0$；

变换矩阵 $\boldsymbol{T}$ 中右上角的 2×1 子矩阵，即 p、q，可使图形产生透视变换，在作其他变换时需使 $p=0$，$q=0$；

变换矩阵 $\boldsymbol{T}$ 中右下角的 s 若不为 1，可使图形产生全比例（x、y 向等比例放大或者缩小）变换。

下面是统一的齐次变换矩阵的参考函数代码：

```
/*************************************************************************
    Get2DMatrix: 创建二维图形的齐次基本变换矩阵
    matrix[][3]: 创建的二维基本变换矩阵; iFlag: 变换矩阵类型 0 - 平移,1 - 旋转,2 - x 镜像,
3 - y 镜像,4 - x 错移,5 - y 错移,6 - 比例; rotateAngle: 旋转角度; x_dis,double y_dis: 平移变换
的距离及错移变换时坐标轴方向系数; dbl_zoom: 全比例缩放系数
*************************************************************************/
void Get2DMatrix(double matrix[][3],int iFlag,double rotateAngle,double x_dis,double y_dis,
double dbl_zoom){
  for(int i = 0;i < 3;i++){//首先创建单位阵
        for(int j = 0;j < 3;j++)
            matrix[i][j] = 0;
    }
    matrix[0][0] = 1;
    matrix[1][1] = 1;
    matrix[2][2] = 1;
    if(iFlag == 0){//平移变换矩阵
        matrix[2][0] = x_dis;
        matrix[2][1] = y_dis;
    }
    else if(iFlag == 1){//旋转变换矩阵
        matrix[0][0] = cos(PI/180 * rotateAngle);
        matrix[0][1] = sin(PI/180 * rotateAngle);
        matrix[1][0] = ( - 1) * sin(PI/180 * rotateAngle);
        matrix[1][1] = cos(PI/180 * rotateAngle);
    }
    else if(iFlag == 2)                               //x 轴镜像
        matrix[1][1] = - 1;
    else if(iFlag == 3)                               //y 轴镜像变换矩阵
        matrix[0][0] = - 1;
    else if(iFlag == 4)                               //x_错移变换矩阵
        matrix[1][0] = x_dis;
    else if(iFlag == 5)                               //y_错移变换矩阵
        matrix[0][1] = y_dis;
    else if(iFlag == 6)                               //全比例变换矩阵
```

```
        matrix[2][2] = dbl_zoom;
}
```

代码中的 PI 表示圆周率 π,使用前,须在文件中先定义该常量:

```
#define PI 3.1415926
```

获得坐标变换矩阵后,用点乘以变换矩阵即得变换后的点,函数参考代码为:

```
/*****************************************************************************
    GetNewPoint: 二维图形点几何变换
    m_point_Array: 变换前及变换后的图形点集合 m_Matrix[][3]: 变换矩阵
*****************************************************************************/
void GetNewPoint(CArray<CPoint,CPoint> m_point_Array,double m_Matrix[][3]){
    CArray<CPoint,CPoint> m_point_Array_new;
    CPoint point_new;
    double s_dbl;
    for(int i = 0;i < m_point_Array.GetSize();i++){
point_new.x = m_point_Array.GetAt(i).x * m_Matrix[0][0] + m_point_Array.GetAt(i).y * m_Matrix
[1][0] + m_Matrix[2][0];
    point_new.y = m_point_Array.GetAt(i).x * m_Matrix[0][1] + m_point_Array.GetAt(i).y * m_
Matrix[1][1] + m_Matrix[2][1];
            s_dbl = m_Matrix[0][2] + m_Matrix[1][2] + m_Matrix[2][2];
            point_new.x/ = s_dbl;
            point_new.y/ = s_dbl;
            m_point_Array_new.Add(point_new);
    }
    m_point_Array.RemoveAll();
    m_point_Array.Append(m_point_Array_new);
}
```

5.2.3 二维组合变换

二维图形的基本变换是相对坐标原点或者坐标轴的图形变换,那么,如何实现相对于平面上任意点或任意直线的变换?这种对任意位置的变换称为复杂几何变换。

实现复杂变换的思路是,将其变换为基本变换的位置,使相对任意点或者直线的变换转换为相对坐标原点或者坐标轴的变换,然后,再按反向顺序返回原任意点或者直线的变换。在实现上述步骤时,需要对图形进行连续多次基本变换,这种由多个基本变换组成复杂变换的方法叫作组合变换或基本变换的级联,有时又称复合变换,相应的矩阵称为组合变换矩阵或基本变换矩阵的级联矩阵。

组合变换中会用到矩阵相乘,其函数代码参考如下:

```
/*****************************************************************************
    MatrixXMatrixOf2D: 两个二维矩阵相乘的函数
    matrix0[][3]: 矩阵 1 及返回的矩阵; matrix1[][3]: 矩阵 2
*****************************************************************************/
    void MatrixXMatrixFor2D(double matrix0[][3],double matrix1[][3]){
        double matrix2[3][3];
        for(int i = 0;i < 3;i++){
```

```
            for(int j = 0;j < 3;j++){
        matrix2[i][j] = matrix0[i][0] * matrix1[0][j] + matrix0[i][1] * matrix1[1][j] + matrix0
    [i][2] * matrix1[2][j];}
        }
            for(i = 0;i < 3;i++){
                for(int j = 0;j < 3;j++){
                    matrix0[i][j] = matrix2[i][j];
                }
            }
        }
```

1. 绕平面上任一点的旋转变换

对于矩形 $ABCD$，要求其绕顶点 $A(x_0,y_0)$旋转 θ 角，如图 5.2-12 所示。由于 A 点不是原点，所以不能直接用旋转变换矩阵，可将此变换分解成几个基本变换的级联，通过组合变换实现。

步骤一，将图形平移，使点 A 与原点重合，成为矩形 $A_1B_1C_1D_1$，变换矩阵为 $\boldsymbol{T}_1$：

$$\boldsymbol{T}_1=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ -x_0 & -y_0 & 1\end{bmatrix}$$

如图 5.2-13 所示。

步骤二，将矩形 $A_1B_1C_1D_1$ 绕 A_1 点(即原点)旋转 θ 角，成为矩形 $A_2B_2C_2D_2$，变换矩阵为 $\boldsymbol{T}_2$：

$$\boldsymbol{T}_2=\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}$$

如图 5.2-14 所示。

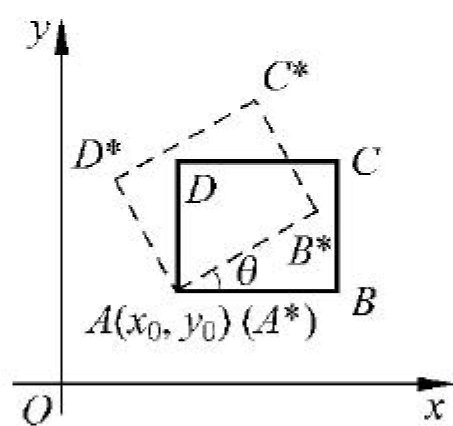


图 5.2-12 绕任意一点旋转

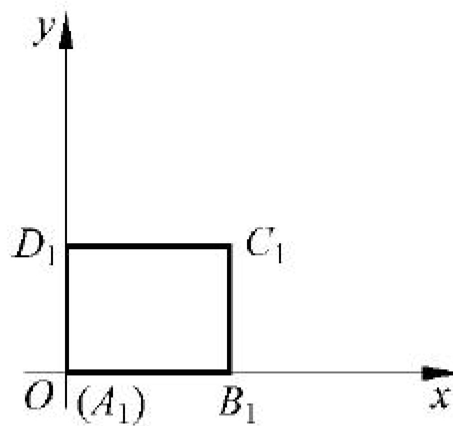


图 5.2-13 平移到坐标原点

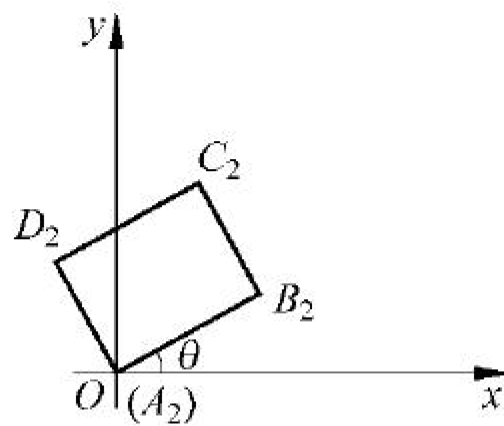


图 5.2-14 在坐标原点旋转

步骤三，将矩形 $A_2B_2C_2D_2$ 平移，使 A 点(即 A_2)回到原来的位置，成为矩形 $A^*B^*C^*D^*$，变换矩阵为 $\boldsymbol{T}_3$：

$$\boldsymbol{T}_3=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ x_0 & y_0 & 1\end{bmatrix}$$

将以上三个变换矩阵相乘，即可得到绕任意点旋转的组合变换矩阵 $\boldsymbol{T}$：

$$\boldsymbol{T}=\boldsymbol{T}_1\boldsymbol{T}_2\boldsymbol{T}_3=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ -x_0 & -y_0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ x_0 & y_0 & 1\end{bmatrix}$$

则绕任意点的旋转变换为

$$P^* = PT = P\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\-x_0 & -y_0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta & \sin\theta & 0\\-\sin\theta & \cos\theta & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\x_0 & y_0 & 1\end{bmatrix}$$

注意：因为矩阵乘法不符合交换律，即对矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，一般 $\boldsymbol{AB}\neq\boldsymbol{BA}$，所以级联的顺序不能颠倒。

计算绕任意点的旋转变换矩阵的函数代码参考如下：

```
/*********************************************************************
RotateTransform2D: 计算绕任意点的旋转变换矩阵的函数
matrix[][3]: 旋转变换矩阵; m_x: 点的 x 坐标; y: 点的 y 坐标; m_Angle: 旋转角度
*********************************************************************/
void RotateTransform2D(double m_Matrix[][3],double& m_x,double& m_y,double& m_Angle){
    double m_Matrix0[3][3];
    Get2DMatrix(m_Matrix,0,0,(-1)*m_x,(-1)*m_y,0); //先平移到坐标原点
    Get2DMatrix(m_Matrix0,1,(-1)*m_Angle,0,0,0);   //旋转 - m_angel
    MatrixXMatrixFor2D(m_Matrix,m_Matrix0);        //矩阵相乘
    Get2DMatrix(m_Matrix0,0,0,m_x,m_y,0);          //从坐标原点平移走
    MatrixXMatrixFor2D(m_Matrix,m_Matrix0);        //矩阵相乘
    return;
}
```

2. 对任意直线的镜像变换

点 $P(x,y)$ 对平面上的任意直线 $y=kx+b$ 进行镜像变换，如图 5.2-15 所示，可将此变换分解成几个基本变换的级联组合。

步骤一，将点连同直线平移，使直线 $y=kx+b$ 与 y 轴的交点 $A(0,b)$ 平移到原点，变换矩阵为 $\boldsymbol{T}_1$：

$$\boldsymbol{T}_1 = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & -b & 1\end{bmatrix}$$

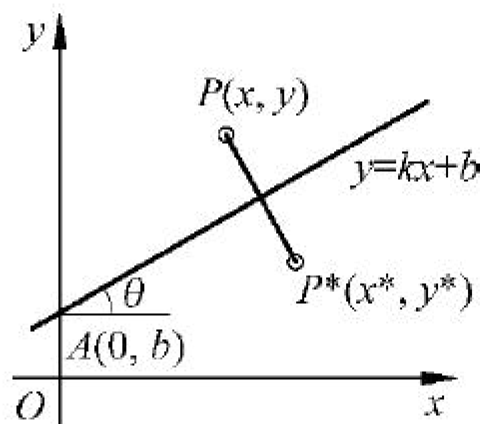


图 5.2-15 对任意直线的镜像变换

步骤二，绕坐标原点旋转 $-\theta$ 角，使直线与 x 轴重合，θ 为直线与 x 轴的夹角，$\theta=\arctan k$，变换矩阵为 $\boldsymbol{T}_2$：

$$\boldsymbol{T}_2 = \begin{bmatrix}\cos(-\theta) & \sin(-\theta) & 0\\-\sin(-\theta) & \cos(-\theta) & 0\\0 & 0 & 1\end{bmatrix}$$

步骤三，对 x 轴进行镜像变换，变换矩阵为 $\boldsymbol{T}_3$：

$$\boldsymbol{T}_3 = \begin{bmatrix}1 & 0 & 0\\0 & -1 & 0\\0 & 0 & 1\end{bmatrix}$$

步骤四，返回原位置(顺序与上相反，即先旋转再平移)。

(1) 绕原点旋转 θ 角，变换矩阵为 $\boldsymbol{T}_4$；

(2) 再平移到原位置，变换矩阵为 $\boldsymbol{T}_5$。

$$T_4=\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix},\quad T_5=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & b & 1\end{bmatrix}$$

将以上五个变换矩阵相乘，即可得到对任意直线镜像的组合变换矩阵：

$$T=T_1T_2T_3T_4T_5=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -b & 1\end{bmatrix}\begin{bmatrix}\cos(-\theta) & \sin(-\theta) & 0\\ -\sin(-\theta) & \cos(-\theta) & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & b & 1\end{bmatrix}$$

点的组合变换为

$$P^*=PT=PT_1T_2T_3T_4T_5$$

计算任意直线的镜像变换矩阵函数代码参考如下：

```
/*******************************************************************
MirrorTransform2D：任意直线的镜像变换矩阵函数
matrix[][3]：返回的最终变换矩阵；m_x0,m_y0：点 1 的坐标；m_x1,m_y1：点 2 的坐标
*******************************************************************/
void MirrorTransform2D(double m_Matrix[][3],double m_x0,double m_y0,double m_x1,double m_y1){
    double m_Matrix0[3][3];
    //区分特殊情况,如果是水平线或者垂直线
    if(m_y1 == m_y0){//水平线
        Get2DMatrix(m_Matrix,0,0,0,(-1)*m_y0,0);       //1.先平移到和 x 轴重合
        Get2DMatrix(m_Matrix0,2,0,0,0,0);              //2.镜像
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);        //矩阵相乘
        Get2DMatrix(m_Matrix0,0,0,0,m_y0,0);           //再反向平移
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);        //矩阵相乘
    }
    else if(m_x1 == m_x0){                             //垂直线
        Get2DMatrix(m_Matrix,0,0,(-1)*m_x0,0,0);       //1.先平移到和 y 轴重合
        Get2DMatrix(m_Matrix0,3,0,0,0,0);              //2.镜像
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);        //矩阵相乘
        Get2DMatrix(m_Matrix0,0,0,(-1)*m_x0,0,0);      //再反向平移
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);        //矩阵相乘
    }
    else{//一般位置直线
        //求交点
        int a, b,c;
        double angle,y_ledge;
        a = m_y0 - m_y1;
        b = m_x1 - m_x0;
        c = m_x0 * m_y1 - m_x1 * m_y0;
        y_ledge = (-1) * c/(double)b;
        angle = atan((-1) * (double(a))/(double)b);    //反正切
        angle = angle * 180/PI;
        Get2DMatrix(m_Matrix,0,0,0,(-1)*y_ledge,0);    //先平移到坐标原点
        Get2DMatrix(m_Matrix0,1,(-1)*angle,0,0,0);     //旋转 - angel
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);
```

```
        Get2DMatrix(m_Matrix0,2,0,0,0,0);          //镜像 - x
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);
        Get2DMatrix(m_Matrix0,1,angle,0,0,0);      //旋转 - angel
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);
        Get2DMatrix(m_Matrix0,0,0,0,y_ledge,0);    //反向平移
        MatrixXMatrixFor2D(m_Matrix,m_Matrix0);
    }
    return;
}
```

3. 任意平面图形的比例变换

对于矩形 $ABCD$,要求相对任意一点如其顶点 $A(x_0, y_0)$等比例放大,如图 5.2-16 所示,可分解成几个基本变换的步骤。

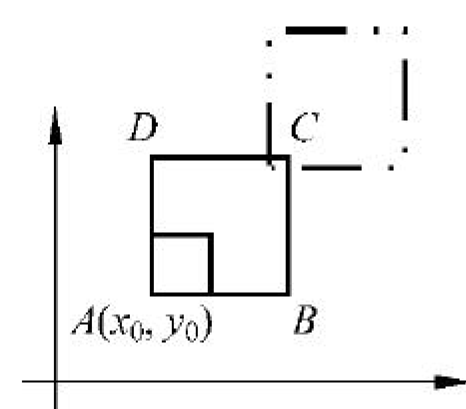


图 5.2-16 比例放大

步骤一,将矩形平移到坐标原点,变换矩阵为 $\boldsymbol{T}_1$:

$$\boldsymbol{T}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix}$$

步骤二,图形等比例放大,变换矩阵为 $\boldsymbol{T}_2$:

$$\boldsymbol{T}_2 = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

步骤三,将变换后的图形反方向平移到原位置,变换矩阵为 $\boldsymbol{T}_3$:

$$\boldsymbol{T}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_0 & y_0 & 1 \end{bmatrix}$$

组合变换矩阵为

$$\boldsymbol{T} = \boldsymbol{T}_1\boldsymbol{T}_2\boldsymbol{T}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix}\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_0 & y_0 & 1 \end{bmatrix}$$

点的组合变换为

$$\boldsymbol{P}^* = \boldsymbol{PT} = \boldsymbol{PT}_1\boldsymbol{T}_2\boldsymbol{T}_3$$

计算绕任意点的全比例变换矩阵函数代码参考如下:

```
/*********************************************************************
ScaleTransform2D: 任意点的全比例变换矩阵
matrix[][3]: 返回的最终变换矩阵; m_x,m_y: 点的坐标; m_scale: 比例系数
/*********************************************************************
void ScaleTransform2D(double m_Matrix[][3],double m_x,double m_y,double m_scale){
    double m_Matrix0[3][3];
    Get2DMatrix(m_Matrix,0,0,(-1)*m_x,(-1)*m_y,0); //1.首先移动到坐标原点
    Get2DMatrix(m_Matrix0,6,0,0,0,m_scale);        //2.比例变换
    MatrixXMatrixFor2D(m_Matrix,m_Matrix0);
    Get2DMatrix(m_Matrix0,0,0,m_x,m_y,0);          //3.反向平移
    MatrixXMatrixFor2D(m_Matrix,m_Matrix0);
}
```

5.2.4 交互技术实现图形变换

无论是基本变换还是组合变换，图形变换的步骤都是相同的：利用变换前的图形点乘以变换矩阵，获得变换后的图形点，从而构成变换后的图形。在实现图形变换时，采用相关的交互操作技术可以产生实时直观的图形变换效果，例如，利用鼠标拖动实现图形变换、只选取特定图形进行变换、把图形的某个端点或者直线中点定位为旋转中心等。交互技术也是计算机图形学的重要研究内容，图形交互是交互技术的主要部分，第 2 章讲述的单击消息函数和非模式对话框的使用以及 3.1.5 节的橡皮筋技术均可实现一定的图形交互功能。在图形变换中，还会用到下面典型的图形交互方法：图形拾取技术、对象捕捉技术、鼠标交互操作技术等。

1. 图形拾取技术

在对屏幕上的图元进行操作时，如果只对其中某个或者几个图元操作，就会用到图形拾取技术，或者称图形选择技术。图形拾取技术是个重要的图形交互手段，不仅可用于图形变换，而且广泛应用于对图形的其他处理，例如，对拾取图形进行修改、删除、分析等。单一图元拾取可用“点选择”方法，即用户在屏幕上交互拾取一点，系统判断该点在哪个图元上，并将该图元用高亮度显示。如果要选择一系列图元，可用“窗口选择”，用光标拖动获得一个矩形选择窗口，将位于该选择窗口内的图元作为被选择的对象。

在进行图形拾取时，需要对所有图元进行判断计算，如果当前处理场景的图元数量相当庞大，对每一个图元都实施精确的判断计算，将使系统反应迟缓。为了提高拾取的效率，可以采取多种预处理的方法。例如，在使用“点选择”方法拾取时，先粗略判断拾取点是否在某图元附近，如不在，则不再对该图元进行进一步判断；如在，则再用较精确的算法进一步判断。

在粗略判断拾取点是否在某图元附近时，一种最简单的方法是判断拾取点是否在包围该图元的最小外接矩形(即边界盒)内。例如，判断拾取点是否在直线 P_1P_2 上，直线两个端点的坐标为 $P_1(x_1,y_1)$、$P_2(x_2,y_2)$，则最小外接矩形为 x 方向在 x_1、x_2 之间和 y 方向在 y_1、y_2 之间的区域。当图元比较复杂时，例如任意位置的圆弧段，可以将圆弧段细分多段后，利用多个最小外接矩形来粗略判断。

粗略判断后，如果拾取点在图元的最小外接矩形内，则需进一步精确判断拾取点是否在图元上。由于拾取点是屏幕的离散像素点，而图元本身是连续的图形，所以精确判断的是拾取点和图元上点的距离是否在精度内，如在精度内，则认为拾取的是该图元，否则对该图元不予拾取。例如，精确判断直线是否被拾取时，根据直线隐式方程的含义，将拾取点代入直线隐式方程，计算 $|ax+by+c|<\varepsilon$，式中 ε 是精度变量，如满足，则该直线被拾取。同理，在判断圆是否被拾取时，可利用圆的方程判断 $|x^2+y^2-R^2|<\varepsilon^2$，椭圆拾取的精确判断方法和圆的判断方法类似。

对于图元是由一组基本图形组合而成的情况，例如，由多条首尾相接的线段构成的多边形图元和裁剪矩形窗口图元，由于多个基本图形组成的是一个整体，图形变换是整体变换，所以，要拾取这个整体图元，而非其中的一个基本图形，当拾取到其中的一个基本图形后，要把这个基本图形所属的图元作为拾取对象。

对于拾取的图形，如果通过图形处理形成了另外一种类型的图形，为了避免后续拾取错误，应在原拾取图形类型的图形集合中删除拾取的图形。例如，单一的一条直线经过图形处理成为了多边形的一条边，那么，在直线集合中删除原拾取的直线。

由于被拾取的图形可能是直线、圆、椭圆、多边形、矩形等图形类型，每种类型的数据结构是不同的，而且，为了突出拾取的图形，对其采用高亮颜色显示，为了方便表示拾取的图形，可以建立图形拾取类结构。对于单个图形的拾取类结构，可参考如下：

```
class CPicker:CDraw{
public:
    CPicker(){
        picktype = pick_none;
        stateFlag = 0;
    }
    Picktype picktype;              //拾取的图元类型
    int stateFlag;                  ///拾取是否变化,0: 不用重画,1: 发生改变须重画
    //可拾取的图元类
    CLine m_Line;                   //拾取的直线
    CCircle m_Circle;               //拾取的圆及圆弧
    CEllipse m_Ellipse;             //拾取的椭圆
    CPolyLine m_PolyLine;           //拾取的多边形
    CCutRect m_CutRect;             //拾取的矩形窗口
};
```

其中，picktype 是拾取的图元类型，定义为枚举结构，可枚举的类型为：

```
enum Picktype { pick_none,pick_line,pick_circle,pick_ellipse,pick_polyline,pick_rect};
```

在“点选择”方法中，在显示器屏幕上用鼠标拾取一个像素点，然后，判断该点是否在某一个图元上，如是，则拾取该图元。由于有多种图元类型，所以需要分别判断。对每个图元的判断方法都是先粗略判断，再精确判断。直线段的拾取判断代码参考如下：

```
/**************************************************************************
    CheckIsPicked: 拾取判断函数
    point: 拾取点; line: 要判断的直线段
**************************************************************************/
     bool CheckIsPicked(CPoint &point,CLine &line) {
     //1.判断是否在图形的最小外接矩形(即边界盒)内
     if(CheckIsInBox(point,line.pt1.x,line.pt2.x,line.pt1.y,line.pt2.y) == false) return false;
     //判断点是否在直线上
    int a = line.pt1.y - line.pt2.y;
    int b = line.pt2.x - line.pt1.x;
    int c = line.pt1.x * line.pt2.y - line.pt2.x * line.pt1.y;
    if(abs(a * point.x + b * point.y + c)> EPSLON) return false;
    else
     return true; }
```

其中，判断是否在图形的最小外接矩形内的函数参考代码如下：

```
bool CheckIsInBox(CPoint &pt,int x1,int x2,int y1,int y2){
    if((pt.x - x1) * (pt.x - x2)> 0)
```

```
        return false;                //在左右边界外
    else if((pt.y - y1) * (pt.y - y2)> 0)
        return false;                //在上下边界外
    else
        return true;
    }
```

EPSLON 为定义的一个精度值,如#define EPSLON 500.0。

对线段组的拾取判断代码如下:

```
/ *******************************************************************
    CheckIsPicked: 拾取判断函数
    point: 拾取点; m_line_array: 要判断的直线段组; m_Picker: 拾取图形
******************************************************************* /
    bool CheckIsPicked(CPoint &point,CArray<CLine,CLine> &m_line_array,CPicker &m_Picker){
        if(m_line_array.GetSize()> 0){
            //逐条线判断
            CLine line;
            for(int i = 0;i < m_line_array.GetSize();i++){
                line = m_line_array.GetAt(i);
                //判断直线是否被拾取
                if(CheckIsPicked(point,line) == true){//有拾取
                    //首先判断是否原拾取,如是,则设置状态 = 1 表明已拾取,且在显示,不必再
//显示,否则先利用异或画原图形,再拾取新图形
    if(m_Picker.picktype == pick_line&&(m_Picker.m_Line.pt1 == line.pt1&&m_Picker.m_Line.
pt2 == line.pt2))
                 m_Picker.stateFlag = 0;
                else{//重新获得拾取的对象
                    m_Picker.stateFlag = 1;
                    m_Picker.picktype = pick_line;
                    m_Picker.m_Line = line;
                    }
                    return true;
                }
            }
        }
        return false;
    }
```

对圆、圆弧段以及椭圆等图元的拾取判断方法和直线的拾取判断方法类似,限于篇幅,本书不再赘述。需要注意的是,对于任意圆弧段的拾取判断,需要进一步细分圆弧段,并对每个细分圆弧段进行最小外接矩形判断。对于由基本图形组合而成的图元,只要判断拾取点在其中一个基本图形上,即拾取整个图元。多边形的拾取判断代码参考如下:

```
/ *******************************************************************
    CheckIsPicked: 拾取判断函数
    point: 拾取点; PolyLine: 要判断的多边形; m_Picker: 拾取图形
/ *******************************************************************
    bool CheckIsPicked(CPoint &point,CPolyLine &PolyLine,CPicker &m_Picker){
        //判断是否在多边形边上
```

```
        int pFlag = 0;                    //是否拾取
        CLine line;
        pFlag = 0;
        for(int j = 0;j < PolyLine.m_PolyLine_array_Out.GetSize();j++){//外环
            line = PolyLine.m_PolyLine_array_Out.GetAt(j);
            if(CheckIsPicked(point,line) == true){
                pFlag = 1;
                break;
            }
            else
                continue;                 //继续判断外环其他边
            }
        if(pFlag == 0){//判断是否在内环
            for(int k = 0;k < PolyLine.in_num;k++){
                for(int j = 0;j < PolyLine.m_PolyLine_array_in[k].GetSize();j++){
                    line = PolyLine.m_PolyLine_array_in[k].GetAt(j);
                    if(CheckIsPicked(point,line) == true){
                        pFlag = 1;
                        break;
                    }
                    else
                        continue;   //继续判断内环其他边
                }
            }
        }
        if(pFlag == 1){//有拾取,首先判断是否原拾取,如是,则设置状态 = 1 表明已拾取,且在显
示,则不必再刷新,否则拾取新图形
    if(m_Picker.picktype == pick_polyline&&(m_Picker.m_PolyLine.m_PolyLine_array_Out.GetAt
(0).pt1 == PolyLine.m_PolyLine_array_Out.GetAt(0).pt1&&m_Picker.m_PolyLine.m_PolyLine_
array_Out.GetAt(0).pt2 == PolyLine.m_PolyLine_array_Out.GetAt(0).pt2&&m_Picker.m_PolyLine.
m_PolyLine_array_Out.GetAt(1).pt2 == PolyLine.m_PolyLine_array_Out.GetAt(1).pt2))
                m_Picker.stateFlag = 0;
            else{//重新获得拾取的对象
                m_Picker.stateFlag = 1;
                m_Picker.picktype = pick_polyline;
                //该多边形加入拾取集合中
                m_Picker.m_PolyLine.m_PolyLine_array_Out.RemoveAll();
m_Picker.m_PolyLine.m_PolyLine_array_Out.Append(PolyLine.m_PolyLine_array_Out);
                m_Picker.m_PolyLine.in_num = PolyLine.in_num;
                for(int k = 0;k < PolyLine.in_num;k++){
                    m_Picker.m_PolyLine.m_PolyLine_array_in[k].RemoveAll();
m_Picker.m_PolyLine.m_PolyLine_array_in[k].Append(PolyLine.m_PolyLine_array_in[k]); }
            }
            return true;
        }
        return false;
    }
```

多边形数组的拾取判断代码如下：

```
bool CheckIsPicked(CPoint &point,CPolyLine * m_PolyLine,int &PolyLine_num,CPicker &m_Picker)
```

```
{//PolyLine_num: 多边形的个数
    for(int i = 0;i < PolyLine_num;i++){
        if(CheckIsPicked(point,m_PolyLine[i],m_Picker) == true)
            return true;
    }
    return false;
}
```

为了对拾取的图形用高亮的色彩显示，在 OnDraw()函数中，需要调用绘制拾取图形的函数，参考代码如下：

```
/********************************************************************
    DrawPicker: 绘制拾取的图形
    pDC: 显示器设备指针; m_Picker: 拾取的图形; crColor: 绘制颜色; ineWidth: 线宽
*********************************************************************/
    void DrawPicker(CDC * pDC,CPicker &m_Picker,COLORREF crColor,int lineWidth = 0){
                                            //画拾取的图形
        //1.判断是否为线段组
        if(m_Picker.picktype == pick_line)          //画直线
    MIDPOINT_Line(pDC,m_Picker.m_Line.pt1,m_Picker.m_Line.pt2,crColor,lineWidth);
        else if(m_Picker.picktype == pick_circle){//2.判断是否为圆
            //画拾取圆
            if(m_Picker.m_Circle.Type == 0)         //整圆
    Mid_Circle(pDC,m_Picker.m_Circle.Opt,m_Picker.m_Circle.rLength,crColor);
            else                                    //是圆弧段
    DrawCircleArc(pDC,m_Picker.m_Circle.Opt,m_Picker.m_Circle.rLength,m_Picker.m_Circle.
Spt,m_Picker.m_Circle.Ept,crColor);
        }
        else if(m_Picker.picktype == pick_ellipse){//3.判断是否为椭圆,画拾取的椭圆
    MidPt_Ellipse(pDC,m_Picker.m_Ellipse.Opt,m_Picker.m_Ellipse.a,m_Picker.m_Ellipse.b,
crColor);
        }
        else if(m_Picker.picktype == pick_polyline){//4.判断是否为多边形,画拾取的多边形
            CLine line;
            //画外环
            for(int j = 0;j < m_Picker.m_PolyLine.m_PolyLine_array_Out.GetSize();j++){
                line = m_Picker.m_PolyLine.m_PolyLine_array_Out.GetAt(j);
                MIDPOINT_Line(pDC,line.pt1,line.pt2,crColor,lineWidth);
            }
            {//画内环
                for(int k = 0;k < m_Picker.m_PolyLine.in_num;k++){
            for(int j = 0;j < m_Picker.m_PolyLine.m_PolyLine_array_in[k].GetSize();j++){
                        line = m_Picker.m_PolyLine.m_PolyLine_array_in[k].GetAt(j);
                        MIDPOINT_Line(pDC,line.pt1,line.pt2,crColor,lineWidth);
                    }
                }
            }
        }
        else if(m_Picker.picktype == pick_rect){//5.判断是否在裁剪矩形上,画拾取的矩形
    MIDPOINT_Line(pDC,CPoint(m_Picker.m_CutRect.xL,m_Picker.m_CutRect.yT),CPoint(m_
Picker.m_CutRect.xL,m_Picker.m_CutRect.yB),crColor,lineWidth);
```

```
    MIDPOINT_Line(pDC,CPoint(m_Picker.m_CutRect.xL,m_Picker.m_CutRect.yT),CPoint(m_
Picker.m_CutRect.xR,m_Picker.m_CutRect.yT),crColor,lineWidth);
    MIDPOINT_Line(pDC,CPoint(m_Picker.m_CutRect.xR,m_Picker.m_CutRect.yB),CPoint(m_
Picker.m_CutRect.xR,m_Picker.m_CutRect.yT),crColor,lineWidth);
    MIDPOINT_Line(pDC,CPoint(m_Picker.m_CutRect.xR,m_Picker.m_CutRect.yB),CPoint(m_
Picker.m_CutRect.xL,m_Picker.m_CutRect.yB),crColor,lineWidth);
        }
    }
```

对拾取的图元进行图形变换的代码如下：

```
/********************************************************************
    TransforOf_2D_Picker：对拾取的图形进行几何变换
    m_Picker：拾取的图形；m_Matrix[][3]：几何变换矩阵
/********************************************************************
    void TransforOf_2D_Picker(CPicker& m_Picker,double m_Matrix[][3]){
        if(m_Picker.picktype == pick_line){//1.判断线段组,转换
            GetNewPoint(m_Picker.m_Line.pt1,m_Matrix);
            GetNewPoint(m_Picker.m_Line.pt2,m_Matrix);
        }
        else if(m_Picker.picktype == pick_circle){//2.判断是否为圆,转换圆
            if(m_Picker.m_Circle.Type == 0){
                GetNewPoint(m_Picker.m_Circle.Opt,m_Matrix);
                m_Picker.m_Circle.rLength/ = m_Matrix[2][2];
            }
            else{
                GetNewPoint(m_Picker.m_Circle.Opt,m_Matrix);
                m_Picker.m_Circle.rLength/ = (double)m_Matrix[2][2];
                GetNewPoint(m_Picker.m_Circle.Spt,m_Matrix);
                GetNewPoint(m_Picker.m_Circle.Ept,m_Matrix);
            }
        }
        else if(m_Picker.picktype == pick_ellipse){//3.判断是否为椭圆,转换拾取的椭圆
            GetNewPoint(m_Picker.m_Ellipse.Opt,m_Matrix);
            m_Picker.m_Ellipse.a/ = m_Matrix[2][2];
            m_Picker.m_Ellipse.b/ = m_Matrix[2][2];
        }
        else if(m_Picker.picktype == pick_polyline){//4.判断是否为多边形边,转换
            CLine line;
            //外环
            for(int j = 0;j < m_Picker.m_PolyLine.m_PolyLine_array_Out.GetSize();j++){
                line = m_Picker.m_PolyLine.m_PolyLine_array_Out.GetAt(j);
                GetNewPoint(line.pt1,m_Matrix);
                GetNewPoint(line.pt2,m_Matrix);
                m_Picker.m_PolyLine.m_PolyLine_array_Out.InsertAt(j,line);
                m_Picker.m_PolyLine.m_PolyLine_array_Out.RemoveAt(j + 1);
            }
            for(int k = 0;k < m_Picker.m_PolyLine.in_num;k++){//内环
        for(int j = 0;j < m_Picker.m_PolyLine.m_PolyLine_array_in[k].GetSize();j++){
                    line = m_Picker.m_PolyLine.m_PolyLine_array_in[k].GetAt(j);
                    GetNewPoint(line.pt1,m_Matrix);
```

```
                    GetNewPoint(line.pt2,m_Matrix);
                    m_Picker.m_PolyLine.m_PolyLine_array_in[k].InsertAt(j,line);
                    m_Picker.m_PolyLine.m_PolyLine_array_in[k].RemoveAt(j+1);
                }
            }
        }
        else if(m_Picker.picktype==pick_rect){//5.判断是否为裁剪矩形,转换
            CPoint pt;
            pt.x=m_Picker.m_CutRect.xL;
            pt.y=m_Picker.m_CutRect.yT;
            GetNewPoint(pt,m_Matrix);
            m_Picker.m_CutRect.xL=pt.x;
            m_Picker.m_CutRect.yT=pt.y;
            pt.x=m_Picker.m_CutRect.xR;
            pt.y=m_Picker.m_CutRect.yB;
            GetNewPoint(pt,m_Matrix);
            m_Picker.m_CutRect.xR=pt.x;
            m_Picker.m_CutRect.yB=pt.y;
        }
    }
```

2. 对象捕捉技术

在对图形进行特定操作时,为了实现设定的目标,需要定位图形的某些特征点,如直线/圆弧的端点、中点、圆心以及相切点等,这种定位操作又称为对象捕捉技术,一般情况下对象捕捉的都是点。因为捕捉的对象是相关图元的特征点,所以,对象捕捉首先需要选取图形,然后,再分析选取的图形,并将图形中和当前操作相关的特征点实时地显示出来,以供选取捕捉。所以,图形选取是实现对象捕捉的前提,对象捕捉也可以看作是对选择图形的二次拾取。注意,该选取的图形与拾取处理的图形可以相同,也可以不同。在前述的图形拾取结构类中,可以再加入供对象捕捉的特征点的拾取,参考如下:

```
class CPicker:CDraw{
public:
    CPicker(){
        ...
        capturedFlag=0;
    }
    ...
    //加入捕捉拾取点的集合
    CArray<CPoint,CPoint>m_capture_point;
    int capturedFlag;      //是否已经拾取过该点,0:没有,1:已经拾取过
};
```

同理,在前述的绘制拾取图形函数 DrawPicker()中也需加入对拾取点的绘制,代码如下:

```
void DrawPicker(CDC *pDC,CPicker &m_Picker,COLORREF crColor,int lineWidth=0){//画拾取的图形
...
if(m_Picker.m_capture_point.GetSize()>0){     //画拾取点
        CPoint pt;
        for(int i=0;i<m_Picker.m_capture_point.GetSize();i++){
            pt=m_Picker.m_capture_point.GetAt(i);
```

```
            DrawPoint(pDC,pt,HIGHLIGHTCOLOR);
        }
    }
}
```

在绘制单个点的函数 DrawPoint()中，为了突出拾取或者捕捉的点，除采用高亮度对其显示外，也可以将其用更“大”的点或者明显图线来绘制。用更“大”点绘制的代码参考如下：

```
void DrawPoint(CDC * pDC,CPoint pt,COLORREF crColor){    //突出绘制点 pt
    CPoint pt1 = pt;pDC->SetPixel(pt,crColor);
    pt1.x -= 1; pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.x += 1;pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.y -= 1; pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.y += 1;pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.x -= 1; pt1.y -= 1; pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.x += 1;pt1.y -= 1; pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.x -= 1;pt1.y += 1;pDC->SetPixel(pt1,crColor);
    pt1 = pt;pt1.x += 1;pt1.y += 1;pDC->SetPixel(pt1,crColor);
}
```

对于拾取的图形，再二次拾取其特征点，用于对象捕捉。其中，拾取直线以及获取直线上拾取点的函数代码参考如下：

```
bool CheckIsCapture(CPoint &point,CArray<CLine,CLine> &m_line_array,CPicker &m_Picker){
//判断是否捕捉直线段的端点及中点
    if(m_line_array.GetSize()> 0){//逐条直线判断是否拾取
        CLine line;
        for(int i = 0;i < m_line_array.GetSize();i++){
            line = m_line_array.GetAt(i);
            if(CheckIsPicked(point,line) == true){    //判断直线是否被拾取
            if(m_Picker.m_capture_point.GetSize()>= 2){
if(m_Picker.m_capture_point.GetAt(0) == line.pt1&&m_Picker.m_capture_point.GetAt(2) ==
line.pt2){//已拾取过,不必重新拾取
            m_Picker.capturedFlag = 1;
                    return true;
                }
                }
                //重新获得拾取的点对象
                CPoint pt;
                m_Picker.m_capture_point.RemoveAll();
                m_Picker.capturedFlag = 0;
                pt = line.pt1;                  //端点 1
                m_Picker.m_capture_point.Add(pt);
                pt = line.pt2;                  //端点 2
                m_Picker.m_capture_point.Add(pt);
                pt.x = (line.pt1.x + line.pt2.x)/2; //中点
                pt.y = (line.pt1.y + line.pt2.y)/2;
                m_Picker.m_capture_point.Add(pt);
                return true;
```

```
            }
        }
    }
    return false;
}
```

在其他拾取图元上获取特征点的方法和上述获取拾取直线上点的方法类似,不再赘述。

3. 鼠标交互操作等交互技术

图形拾取、对象捕捉以及实时图形变换等,都离不开鼠标的交互操作。由于鼠标操作非常直观和交互感强,所以在计算机图形系统中,它是一种非常重要的交互手段。鼠标操作有移动鼠标、鼠标左右键单击、鼠标滚轮操作以及这几种鼠标操作的组合等多种。

通过移动鼠标可以拾取和捕捉图形,也可以直接进行图形变换,并利用"异或"绘图特性或者直接刷新屏幕,来实时显示变换的图形。在 VC++ 中,鼠标移动对应的消息函数是 OnMouseMove(),其参数 point 中包含鼠标当前所在屏幕点的坐标信息。利用当前鼠标点和图形进行计算判断,即可实现图形拾取,或者将当前鼠标点和前一个鼠标移动点的坐标进行比较,即可实现平移变换或者缩放判断等。例如,利用鼠标实时图形拾取,在 OnMouseMove()函数中的代码参考如下:

```
if(this->m_iflag==-1){//m_iflag=-1,表示此时没有其他图形功能操作,可以进行拾取
    //逐个对现有的图形判断,是否被拾取,或已被拾取,m_Picker0 为临时拾取图形类
    int iflag=0;                                                  //设置拾取标识
    if(CheckIsPicked(point,m_line_array,m_Picker0)==true)          //判断直线段的拾取
        iflag=1;
    else if(CheckIsPicked(point,m_circle_array,m_Picker0)==true)   //判断圆的拾取
        iflag=1;
    else if(CheckIsPicked(point,m_ellipse_array,m_Picker0)==true)  //判断椭圆的拾取
        iflag=1;
    else if(CheckIsPicked(point,m_PolyLine,iPolyLine,m_Picker0)==true)    //判断多边形
        iflag=1;
    else if(CheckIsPicked(point,m_cutRect,m_Picker0)==true)        //判断裁剪矩形的拾取
        iflag=1;
    if(iflag==1){//有拾取,则判断是否重画
        if(m_Picker0.stateFlag==1){
            Invalidate();
        }
    }
    else{//无拾取,如原来有拾取,则不再显示
        if(m_Picker0.picktype!=pick_none){
            m_Picker0.picktype=pick_none;
            Invalidate();
        }
    }
}
```

单击操作主要起一个确认的作用。例如,在上述鼠标移动拾取某图元后单击,确定拾取操作结束,不再拾取其他图元,单击消息函数 OnLButtonDown()中,完成上述拾取确定的代码参考如下:

```
if(this->m_iFlag==-1){
    if(m_Picker0.picktype!=pick_none){
        CopyPicker(m_Picker,m_Picker0);             //设置临时拾取图形成为正式拾取
        m_Picker0.picktype=pick_none;
        m_Picker0.m_capture_point.RemoveAll();
        Invalidate();
    }
}
```

对象捕捉的鼠标交互操作和图形拾取的鼠标操作方法类似，在鼠标移动消息函数OnMouseMove()中，捕捉选取图形的特征点(代码和上述图形拾取类似，不再赘述)，然后，通过单击对捕捉点进行确认。例如，在旋转变换中，在OnLButtonDown()函数中确定旋转中心的参考代码如下：

```
if(m_iFlag==9){//旋转变换
 m_Transform2DDlg->m_X=point.x;                   //m_Transform2DDlg是旋转变换对话框
 m_Transform2DDlg->m_Y=point.y;
for(int i=0;i<m_Picker0.m_capture_point.GetSize();i++){     //判断当前点是否捕捉的点
if((point.x-m_Picker0.m_capture_point.GetAt(i).x)*(point.x-m_Picker0.m_capture_point.
GetAt(i).x)+(point.y-m_Picker0.m_capture_point.GetAt(i).y)*(point.y-m_Picker0.m_
capture_point.GetAt(i).y)<EPSLON){                 //将捕捉点赋给旋转中心
    m_Transform2DDlg->m_X=m_Picker0.m_capture_point.GetAt(i).x;
    m_Transform2DDlg->m_Y=m_Picker0.m_capture_point.GetAt(i).y;
    m_Picker.m_capture_point.RemoveAll();
    m_Picker.m_capture_point.Add(m_Picker0.m_capture_point.GetAt(i));
    m_Picker0.m_capture_point.RemoveAll();
    Invalidate();                                  //刷新屏幕
    break;
    }
}
m_Transform2DDlg->UpdateData(FALSE);
}
```

将两种鼠标操作组合在一起，也可以实现相关图形处理，例如，鼠标移动的同时按下左键，以实现图形的实时移动、缩放、旋转等几何变换。在鼠标移动函数OnMouseMove()中，图形的平移变换代码参考如下：

```
if(this->m_iFlag==8){                              //m_iFlag=8是平移变换的标识
    if(nFlags==MK_LBUTTON&&pickstep==1){           //是否左键同时被按下
        int x_dis=point.x-pickPt.x;                //计算x方向移动量
        int y_dis=point.y-pickPt.y;                //计算y方向移动量
        double m_Matrix[3][3];
        Get2DMatrix(m_Matrix,0,0,x_dis,y_dis,0);   //移动变换矩阵
        TransforOf2D_Single(m_Matrix);             //对拾取图形进行变换
        pickPt=point;                              //保存当前点
        Invalidate();                              //刷新屏幕
    }
```

利用对话框进行数据交互也是图形处理的常用手段，例如对于旋转变换，旋转角度和旋转中心除了采用鼠标交互外，也使用对话框的形式动态设置变换数据。上述的旋转变换非模式对话框 m_Transform2DDlg 如图 5.2-17 所示。通过微调按钮 Spin 设置旋转角度并调用旋转变换函数。m_Transform2DDlg 中微调按钮消息函数代码参考如下：

```
void CTransform2DDlg::OnDeltaposSpin1(NMHDR * pNMHDR, LRESULT * pResult) {
    NM_UPDOWN * pNMUpDown = (NM_UPDOWN * )pNMHDR;
    UpdateData(TRUE);
    this->m_dblAngle -= 1 * pNMUpDown->iDelta;          //设置显示的旋转角度
    double angle = (-1.0) * pNMUpDown->iDelta;          //每次变换旋转 1°
    m_pView->Rotate2D(angle,this->m_X,this->m_Y);       //在 view 里执行旋转变换
    UpdateData(FALSE);
    *pResult = 0;
}
```

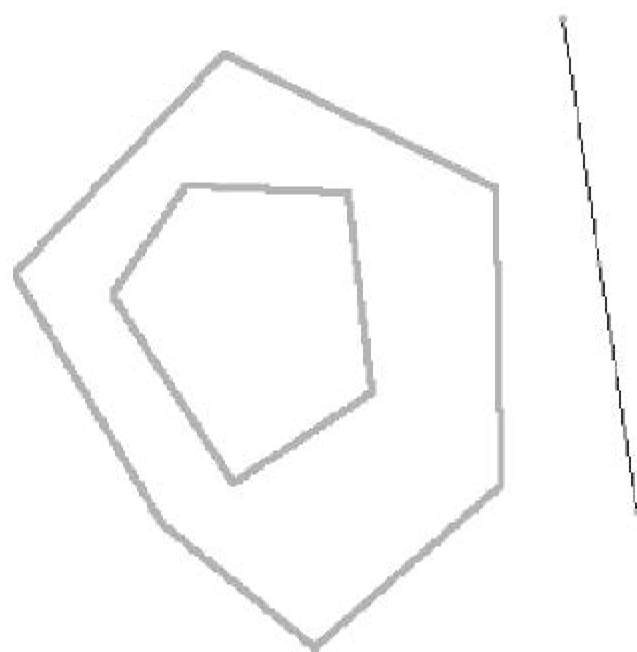




图 5.2-17 利用对话框交互实现旋转变换

视图中的旋转变换函数参考代码为：

```
void CCGTest002View::Rotate2D(double &m_dblAngle,double &m_dblX,double &m_dblY){
double m_Matrix[3][3];
RotateTransform2D(m_Matrix,m_dblX,m_dblY,m_dblAngle); //计算旋转变换矩阵
TransforOf2D_Single(m_Matrix);                        //拾取图形旋转变换
Invalidate();
}
```

在进行图形处理时，只有具备某种条件才能进行某种对应的图形操作。为了避免不满足条件时的误操作，需进行条件判断，并设置是否可执行图形操作功能的状态。例如，对拾取图形的移动变换，只有拾取了图形，移动变换工具栏或者菜单项才处于激活状态，如无拾取，该工具栏或菜单项则处于失效状态，不能进行单击操作。设置方法是处理该工具栏 ID 对应的消息 UPDATE_COMMAND_UI，消息处理函数参考代码如下：

```
void CCGTest002View::OnUpdateMove2d(CCmdUI * pCmdUI) {
    pCmdUI->Enable(m_Picker.picktype!= pick_none?true:false);
}
```

5.3　三维图形几何变换

5.3.1　三维图形基本变换及组合变换

三维图形的几何变换和二维图形的几何变换非常类似，区别在于二维图形是平面图形，只有 x 和 y 坐标；三维图形是立体空间图形，三维图形点除了 x 和 y 坐标外，还有 z 坐标。

设三维空间点 $P(x,y,z)$，用齐次坐标表示为 $[x \quad y \quad z \quad 1]$，三维空间点的几何变换为

$$[x^* \quad y^* \quad z^* \quad 1] = [x \quad y \quad z \quad 1]\boldsymbol{T}$$

其中，$\boldsymbol{T}$ 为三维变换矩阵，具体如下：

$$\boldsymbol{T} = \begin{bmatrix} a & b & c & p \\ d & e & f & q \\ h & i & j & r \\ l & m & n & s \end{bmatrix}$$

在变换矩阵 $\boldsymbol{T}$ 中，$\begin{bmatrix} a & b & c \\ d & e & f \\ h & i & j \end{bmatrix}$使图形产生比例、镜像、错移、旋转等基本变换；$[l \quad m \quad n]$使图形产生平移变换；$[p \quad q \quad r]^{\mathrm{T}}$ 可使图形产生透视变换；$[s]$使图形产生全比例变换。

1. 恒等变换

变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. 比例变换

变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & e & 0 & 0 \\ 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中，a、e、j 分别是 x、y、z 方向的比例因子。

$a=e=j=1$——恒等变换；

$a=e=j>1$——x、y、z 三向等比例放大；

$a=e=j<1$——x、y、z 三向等比例缩小；

$a\neq e\neq j$——x、y、z 三向放大倍数不一样，图形畸变；

$a=0$——图形压缩到 yOz 平面上；

$e=0$——图形压缩到 xOz 平面上；
$j=0$——图形压缩到 xOy 平面上；
$a=0,e=0$——图形压缩到 z 轴上；
$a=0,j=0$——图形压缩到 y 轴上；
$e=0,j=0$——图形压缩到 x 轴上；
$a=0,e=0,j=0$——图形压缩为一点，即原点。

3. 全比例变换

变换矩阵为

$$T=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&s\end{bmatrix}$$

4. 镜像变换

对 xOy 平面的镜像变换矩阵为

$$T=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&0\\0&0&0&s\end{bmatrix}$$

xOz 平面的镜像变换矩阵为

$$T=\begin{bmatrix}1&0&0&0\\0&-1&0&0\\0&0&1&0\\0&0&0&s\end{bmatrix}$$

yOz 平面的镜像变换矩阵为

$$T=\begin{bmatrix}-1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&s\end{bmatrix}$$

5. 错移变换

沿 x 轴错移变换有两种情况。

① 沿 x 含 y 错移（x 坐标的变化量为 dy），变换矩阵为

$$T=\begin{bmatrix}1&0&0&0\\d&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

② 沿 x 含 z 错移（x 坐标的变化量为 hz），变换矩阵为

$$T=\begin{bmatrix}1&0&0&0\\0&1&0&0\\h&0&1&0\\0&0&0&1\end{bmatrix}$$

沿 y 轴错移变换有两种情况。

① 沿 y 含 x 错移(y 坐标的变化量为 bx),变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1 & b & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

② 沿 y 含 z 错移(y 坐标的变化量为 iz),变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & i & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

沿 z 轴错移变换有两种情况。

① 沿 z 含 x 错移(z 坐标的变化量为 cx),变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1 & 0 & c & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

② 沿 z 含 y 错移(z 坐标的变化量为 fy),变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & f & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

6. 旋转变换

在三维空间中,分别可以绕 x、y、z 三个轴进行旋转变换。绕 x、y、z 轴的旋转角分别用 θ、φ、ψ 表示,角度正负按右手定则确定。

根据二维旋转变换可以直接推导得出三维旋转变换,其中,绕 z 轴旋转变换公式为

$$\begin{cases}x^* = x\cos\psi - y\sin\psi\\y^* = x\sin\psi + y\cos\psi\\z^* = z\end{cases}$$

变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}\cos\psi & \sin\psi & 0 & 0\\-\sin\psi & \cos\psi & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

绕 x 轴旋转,可把 y 轴看成 x 轴,z 轴看成 y 轴,符合右手定则,如图 5.3-1 所示。利用二维旋转变换可推导 x 轴旋转变换公式为

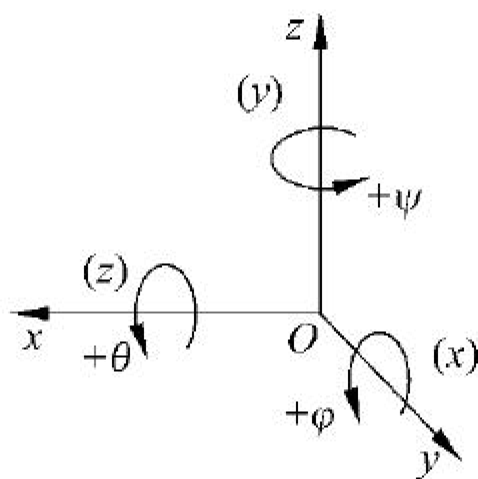


图 5.3-1 绕 x 轴旋转变换

$$\begin{cases} x^* = x \\ y^* = y\cos\theta - z\sin\theta \\ z^* = y\sin\theta + z\cos\theta \end{cases}$$

变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕 y 轴旋转，可把 z 轴看成 x 轴，x 轴看成 y 轴，如图 5.3-2 所示，符合右手定则。利用二维旋转变换可推导绕 y 轴旋转变换公式为

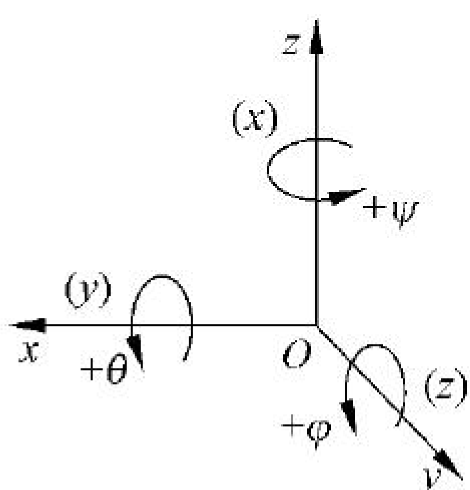


图 5.3-2 绕 y 轴旋转变换

$$\begin{cases} x^* = x\cos\varphi + z\sin\varphi \\ y^* = y \\ z^* = -x\sin\varphi + z\cos\varphi \end{cases}$$

变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

7. 平移变换

变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix}$$

其中，l、m、n 分别是 x、y、z 方向的移动量。

同二维变换，三维变换的基本变换都是相对于原点，或某一坐标轴，或某一坐标平面的，那么相对于任意点、任意直线或任意平面的变换就要用到组合变换。

同二维组合变换，也是先把相对于任意点、任意直线或任意平面的变换先转换为基本变换（即相对于原点或三个坐标轴或三个坐标平面的变换），再顺序返回。

三维图形变换统一的变换矩阵函数代码参考如下：

```
/*************************************************************************
    GetMatrix: 创建三维图形的齐次基本变换矩阵
    matrix[][4]: 创建的二维基本变换矩阵; iFlag:0 移动,1: x 轴旋转,2: y 轴旋转,3: z 轴旋转,
4: x 镜像,5: y 镜像,6: z 镜像,7: x 错移,8: y 错移,9: z 错移,10: 缩放; rotateAngle: 旋转角度;
x_dis,double y_dis,z_dis: 平移变换的距离及错移变换时坐标轴方向系数; dbl_zoom: 全比例缩放
系数
    ***********************************************************************/
    void GetMatrix(double matrix[][4], int iFlag, double x_dis, double y_dis, double z_dis,
double rotateAngle, double dbl_zoom){
        for(int i = 0;i < 4;i++){
            for(int j = 0;j < 4;j++)
```

```
            (matrix[i][j]) = 0;
        }
        matrix[0][0] = 1;matrix[1][1] = 1;matrix[2][2] = 1;matrix[3][3] = 1;
        if(iFlag == 0)                                //移动
            matrix[3][0] = x_dis;matrix[3][1] = y_dis;matrix[3][2] = z_dis;
        else if(iFlag == 1){                          //x 旋转
            matrix[1][1] = cos(PI/180 * rotateAngle);
            matrix[1][2] = sin(PI/180 * rotateAngle);
            matrix[2][1] = (-1) * sin(PI/180 * rotateAngle);
            matrix[2][2] = cos(PI/180 * rotateAngle);
        }
        else if(iFlag == 2){                          //y 旋转
            matrix[0][0] = cos(PI/180 * rotateAngle);
            matrix[0][2] = (-1) * sin(PI/180 * rotateAngle);
            matrix[2][0] = sin(PI/180 * rotateAngle);
            matrix[2][2] = cos(PI/180 * rotateAngle);
        }
        else if(iFlag == 3){                          //z 旋转
            matrix[0][0] = cos(PI/180 * rotateAngle);
            matrix[1][0] = (-1) * sin(PI/180 * rotateAngle);
            matrix[0][1] = sin(PI/180 * rotateAngle);
            matrix[1][1] = cos(PI/180 * rotateAngle);
        }
        else if(iFlag == 4)                           //xOy 镜像
            matrix[2][2] = -1;
        else if(iFlag == 5)                           //yOz 镜像
            matrix[0][0] = -1;
        else if(iFlag == 6                            //zOx 镜像
            matrix[1][1] = -1;
        else if(iFlag == 7){                          //x 错移
            matrix[1][0] = y_dis;
            matrix[2][0] = z_dis;}
        else if(iFlag == 8){                          //y 错移
            matrix[0][1] = x_dis;
            matrix[2][1] = z_dis;
        }
        else if(iFlag == 9){                          //z 错移
            matrix[0][2] = x_dis;
            matrix[1][2] = z_dis;
        }
        else if(iFlag == 10)                          //缩放
            matrix[3][3] = dbl_zoom;
    }
```

三维图形组合变换时,矩阵相乘的函数代码参考如下:

```
/***********************************************************************
    MatrixXMatrix: 矩阵相乘函数
    matrix0[][4]: 矩阵 1 及返回的矩阵; matrix1[][4]: 矩阵 2
***********************************************************************/
    void MatrixXMatrix(double matrix0[][4],double matrix1[][4]){
```

```
        double matrix2[4][4];
        for(int i = 0;i < 4;i++){
            for(int j = 0;j < 4;j++){
                matrix2[i][j] = 0.;
                for(int k = 0;k < 4;k++){             //旧点
                    matrix2[i][j] += matrix0[i][k] * matrix1[k][j];
                }
            }
        }
        for(i = 0;i < 4;i++){
            for(int j = 0;j < 4;j++){
                matrix0[i][j] = matrix2[i][j];
            }
        }
    }
```

5.3.2 三维图形的线框拉伸造型方法

在进行三维图形变换前，首先需要解决三维图形的输入和表示等问题。对于二维平面图形，由于维数和显示设备相同，所以，二维图形点可以直接通过鼠标屏幕拾取获得，而三维图形点是空间立体点，三维点的坐标不能通过鼠标屏幕拾取获得。一种比较简单的三维图形点的产生方法是将拾取的屏幕点增加 z 坐标值并使 $z=0$，从而把二维点转化成三维点。在计算机中，空间形体的生成采用的是由点形成线、线构成面、面构成物体、物体生成场景的方法，因此，对空间物体通常可将其视为某些元素即点、线、面等的集合。三维空间形体在计算机内常用的表示方法有线框模型、表面模型和实体模型，如图 5.3-3 所示。所谓线框模型是用物体的棱边和轮廓线来表示一个物体的几何外貌，这时，整个物体看起来就像建筑物的框架。表面模型是利用组成物体表面的有边界的面集合来表示物体的形状，由于面是由点、线构成的，因此，面模型中包含了物体线框模型的信息。实体模型则既包含了物体表面的信息，又包含了物体内部实心部分的信息，因此实体模型可以完整地描述一个实际的几何物体。

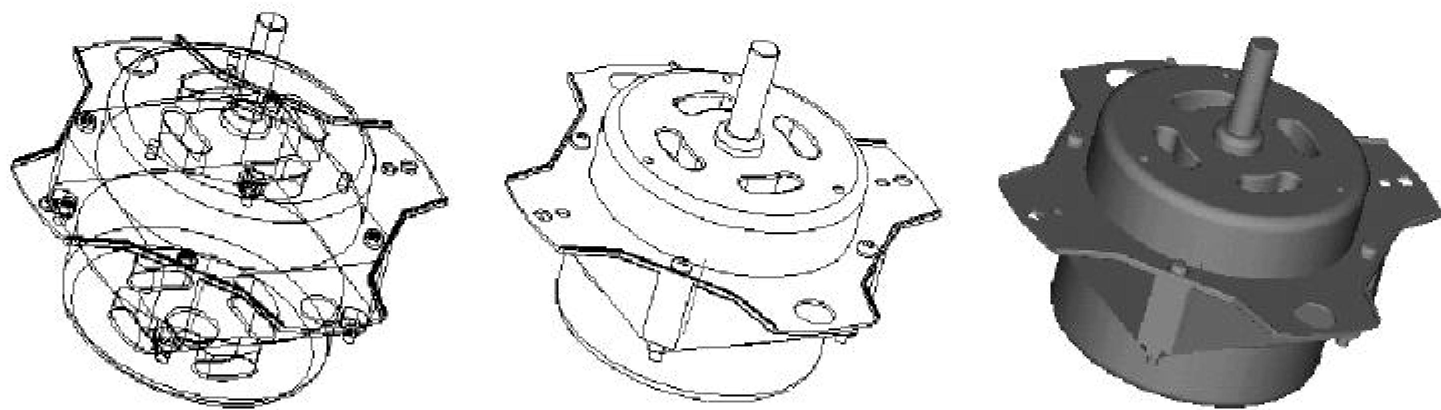


图 5.3-3　三维形体表示：线框模型、表面模型、实体模型

三维形体的常用造型方法有平移扫掠造型和旋转扫掠造型。平移扫掠是指二维平面图形向某一方向扫掠（最常用的是平面垂直方向拉伸）而产生空间形体，旋转扫掠则是指一个二维平面图形围绕一个轴线进行旋转而产生空间形体的造型方法。

在三维图形变换时，由于不涉及复杂的图形处理，因此，三维形体采用较为简单的线框模型表示，并通过将屏幕二维图形沿 z 向拉伸扫掠的方法进行三维造型。

一个简单平面线框拉伸体的结构逻辑可如图 5.3-4 所示。

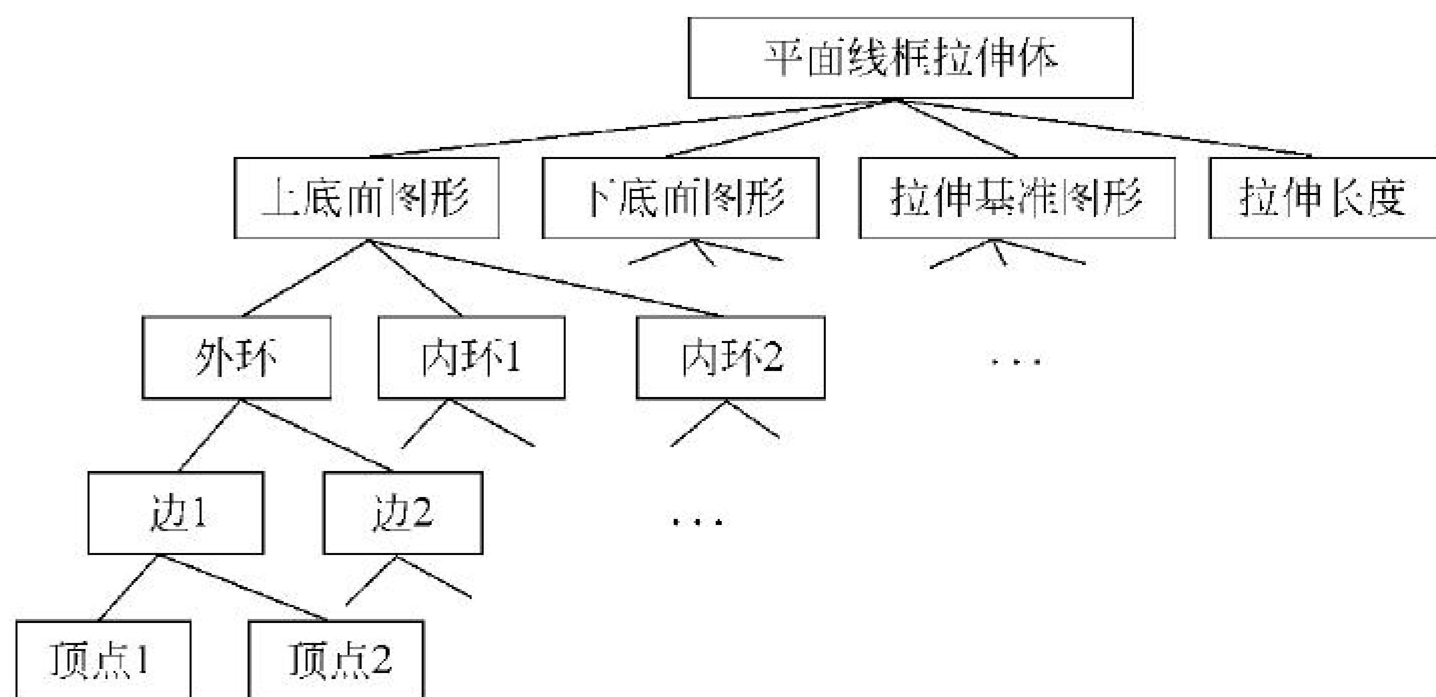


图 5.3-4　简单平面线框拉伸体的结构逻辑

三维图形顶点的齐次坐标表示可参考如下：

```
class CPoint3D{
public:
    CPoint3D(){
        x = 0;y = 0;z = 0;s = 1.0;}
    double x ,y, z,s;                          //三维点的 x、y、z 坐标变量
    CPoint3D& operator = (const CPoint &pt){   //操作符重载,将二维点赋给三维点
        this -> x = pt.x;this -> y = pt.y;this -> z = 0.;
        return * this;
    }
    CPoint3D& operator = (const CPoint3D &pt3){   //操作符重载,三维点赋值
        if(this ==  &pt3){
            return * this;
        }
        else{
            this -> x = pt3.x;this -> y = pt3.y;this -> z = pt3.z;
            return * this;
        }
    }
    bool operator  ==  (const CPoint3D &ps){      //操作符重载,判断两个三维点是否相同
if(abs(this -> x - ps.x)< error&&abs(this -> y - ps.y)< error&&abs(this -> z - ps.z)< error){
            return true;
        }
        else
            return false;
    }
    CPoint& To2DPt(){                          //三维点转化为屏幕二维点,只取点的 x、y 坐标
        CPoint pt;
        pt.x = (int)(this -> x + 0.5);pt.y = (int)(this -> y + 0.5);
        return pt;
    }
};
```

三维形体边的数据结构类的代码参考如下：

```
class CEdge{
public:
    CEdge(){};
    CPoint3D Pt1_3D, Pt2_3D;                    //棱边两个端点
    CEdge& operator = (const CEdge &edge){      //操作符重载,棱边相等
        if(this == &edge)
        {return * this;}
        else{
            this -> Pt1_3D = edge.Pt1_3D;
            this -> Pt2_3D = edge.Pt2_3D;
            return * this;
        }
    }
    CEdge& operator = (const CLine &line){      //操作符重载,棱边等于线段
        this -> Pt1_3D = line.pt1;
        this -> Pt2_3D = line.pt2;
        return * this;
    }
    bool operator == (const CEdge &edge){       //操作符重载,判断棱边是否相同
if((this -> Pt1_3D == edge.Pt1_3D&&this -> Pt2_3D == edge.Pt2_3D)||(this -> Pt1_3D == edge.
Pt2_3D&&this -> Pt2_3D == edge.Pt1_3D)){
            return true;
        }
        else
            return false;
    }
};
```

由封闭的环边组成的平面图形的结构类代码参考如下：

```
class CEdgePlane{
public:
    CEdgePlane(){                               //构造函数
        in_num = 0;
    }
    ~CEdgePlane(){                              //析构函数
        this -> loop_out.RemoveAll();
        for(int i = 0;i < this -> in_num;i++)
            this -> loop_in[i].RemoveAll();
        this -> in_num = 0;
    }
    CEdgePlane& operator = (CEdgePlane &edgeplane){ //操作符重载
        if(this == &edgeplane){
            return * this;}
        else{
            for(int i = 0;i < edgeplane.loop_out.GetSize();i++){
                this -> loop_out.RemoveAll();
                this -> loop_out.Append(edgeplane.loop_out);}
            for(i = 0;i < edgeplane.in_num;i++){
```

```
                this->loop_in[i].RemoveAll();
                this->loop_in[i].Append(edgeplane.loop_in[i]);}
            this->in_num = edgeplane.in_num;
            return *this;
        }
    }
    CArray<CEdge,CEdge> loop_out;                   //外环
    CArray<CEdge,CEdge> loop_in[1000];              //内环数组
    int in_num;                                     //内环数量
};
```

拉伸线框实体的结构类代码参考如下：

```
class CBody_Stretch{
public:
    CBody_Stretch(){length = 0;}
    ~CBody_Stretch(){                               //实体析构函数
        this->polyline.m_PolyLine_array_Out.RemoveAll();
        for(int i = 0;i<polyline.in_num;i++){
            polyline.m_PolyLine_array_in[i].RemoveAll();}
        length = 0;
        for(i = 0;i<2;i++)
            EdgePlane[i].~CEdgePlane();
    }
    void ClearBody(){                               //实体清空函数
        this->polyline.m_PolyLine_array_Out.RemoveAll();
        for(int i = 0;i<polyline.in_num;i++){
            polyline.m_PolyLine_array_in[i].RemoveAll();}
        length = 0;
        for(i = 0;i<2;i++)
            EdgePlane[i].~CEdgePlane();
    }
    CBody_Stretch& operator = (CBody_Stretch &body){//实体复制
        if(this == &body)
            return *this;
        else{
            for(int i = 0;i<2;i++)
                this->EdgePlane[i] = body.EdgePlane[i];
        this->polyline = body.polyline;
        return *this;
        }
    }
    bool operator == (CBody_Stretch &body){         //判断两个实体是否相同
        if(this == &body) return true;
        else if(this->polyline == body.polyline)
            return true;
        else
            return false;
    }
    CEdgePlane EdgePlane[2];                        //拉伸的上下两个平面图形
    double length;                                  //拉伸长度
    CPolyLine polyline;                             //保留最初的拉伸多边形,以便修改
};
```

当给定或者拾取一个平面图形后，如多边形，则通过拉伸可得到立体图形。由于垂直拉伸存在两个方向，为了区分向哪个方向拉伸，需要对平面多边形的方向进行设置。一般设置多边形的外环为逆时针走向，内环为顺时针走向，这样，当拉伸长度为正值时，则沿外环的有向线段叉积的矢量方向(即 z 轴正向)拉伸；当拉伸长度为负值时，沿外环有向线段叉积矢量方向的反向(即 z 轴正向)拉伸。函数代码参考如下：

```
/*********************************************************************
    CreateBodyOfStretch: 创建拉伸立体
    m_Body: 创建的拉伸线框立体; m_Picker: 拾取的平面图形; m_dblLength: 拉伸长度
    *****************************************************************/
    bool CreateBodyOfStretch(CBody_Stretch &m_Body,CPicker& m_Picker,double &m_dblLength)
{//将拾取多边形转化为线框拉伸体的拉伸平面图形
        if(m_Picker.picktype == pick_polyline){
            CBody_Stretch Body;
            CEdge edge,edge1;
            CArray<CEdge,CEdge> m_array_edge,m_array_edge1;
            CPolyLine polyline;                              //临时多边形,用来操作
            polyline = m_Picker.m_PolyLine;
            CheckAndSetDirectionOfPolyline(polyline);        //判断并设置多边形的走向
            CLine line;
            //外环拉伸,构造上下两个表面多边形的外环
            m_array_edge.RemoveAll();m_array_edge1.RemoveAll();
            for(int j = 0;j < polyline.m_PolyLine_array_Out.GetSize();j++){
                line = polyline.m_PolyLine_array_Out.GetAt(j);
                edge1 = edge = line;
                edge1.Pt1_3D.z = m_dblLength;
                edge1.Pt2_3D.z = m_dblLength;
                m_array_edge.Add(edge);
                m_array_edge1.Add(edge1);
            }
            Body.EdgePlane[0].loop_out.Append(m_array_edge);  //加入外环
            Body.EdgePlane[1].loop_out.Append(m_array_edge1); //加入外环
            //内环拉伸,构造上下两个表面多边形的内环
            if(polyline.in_num > 0){
                Body.EdgePlane[0].in_num = polyline.in_num;
                Body.EdgePlane[1].in_num = polyline.in_num;
            }
            for(int k = 0;k < polyline.in_num;k++){
                m_array_edge.RemoveAll();
                m_array_edge1.RemoveAll();
                for(int j = 0;j < polyline.m_PolyLine_array_in[k].GetSize();j++){
                    line = polyline.m_PolyLine_array_in[k].GetAt(j);
                    edge1 = edge = line;
                    edge1.Pt1_3D.z = m_dblLength;
                    edge1.Pt2_3D.z = m_dblLength;
                    m_array_edge.Add(edge);
                    m_array_edge1.Add(edge1);
                }
                Body.EdgePlane[0].loop_in[k].Append(m_array_edge);
                Body.EdgePlane[1].loop_in[k].Append(m_array_edge1);
```

```
            }
            Body.length = m_dblLength;
            Body.polyline = m_Picker.m_PolyLine;
            m_Body = Body;
            return true;
        }
        else
            return false;
    }
```

其中,判断并设置多边形的走向的函数代码如下:

```
void CheckAndSetDirectionOfPolyline(CPolyLine& polyline)
{//判断并设置多边形内外环的方向,外环逆时针,内环顺时针
    CArray<CPoint,CPoint> m_point_Array;
    //首先判断处理外环
    int ringFlag;                                              //内外环标识符
    ringFlag = 1;                                              //1: 外环,0: 内环
    CLine line;
    CArray<CLine,CLine> m_line_array;
    SortForPolyline(polyline.m_PolyLine_array_Out,ringFlag,m_point_Array);
    for(int i = 0;i<m_point_Array.GetSize() - 1;i++){
        line.pt1 = m_point_Array.GetAt(i);
        line.pt2 = m_point_Array.GetAt(i + 1);
        m_line_array.Add(line);
     }
    polyline.m_PolyLine_array_Out.RemoveAll();
    polyline.m_PolyLine_array_Out.Append(m_line_array);
    //再处理内环
    for(i = 0;i<polyline.in_num;i++) {
        ringFlag = 0;
        m_line_array.RemoveAll();
        SortForPolyline(polyline.m_PolyLine_array_in[i],ringFlag,m_point_Array);{
            for(int i = 0;i<m_point_Array.GetSize() - 1;i++){
                line.pt1 = m_point_Array.GetAt(i);
                line.pt2 = m_point_Array.GetAt(i + 1);
                m_line_array.Add(line);
            }
            polyline.m_PolyLine_array_in[i].RemoveAll();
            polyline.m_PolyLine_array_in[i].Append(m_line_array);}
    }
}
```

其中,多边形方向的排序函数SortForPolyline()的代码在4.2.3节中已经列出,此处不再赘述。

有了三维立体图形后,即可实现对该三维图形的几何变换,函数代码如下:

```
/*********************************************************************
    GetNewPoint: 三维图形的几何变换
    m_Body: 几何变换的拉伸线框立体; m_Matrix: 几何变换矩阵
 *********************************************************************/
```

```
void GetNewPoint(CBody_Stretch &m_Body,double m_Matrix[][4]){
    CPoint3D pt0_1,pt1_1,pt0_2,pt1_2;
    CEdge edge;
    for(int i=0;i<m_Body.EdgePlane[0].loop_out.GetSize();i++){    //外环
        pt0_1=m_Body.EdgePlane[0].loop_out.GetAt(i).Pt1_3D;
        pt0_2=m_Body.EdgePlane[0].loop_out.GetAt(i).Pt2_3D;
        GetNewPoint(pt0_1,m_Matrix);
        GetNewPoint(pt0_2,m_Matrix);
        edge.Pt1_3D=pt0_1;
        edge.Pt2_3D=pt0_2;
        m_Body.EdgePlane[0].loop_out.InsertAt(i,edge);
        m_Body.EdgePlane[0].loop_out.RemoveAt(i+1);
        pt1_1=m_Body.EdgePlane[1].loop_out.GetAt(i).Pt1_3D;
        pt1_2=m_Body.EdgePlane[1].loop_out.GetAt(i).Pt2_3D;
        GetNewPoint(pt1_1,m_Matrix);
        GetNewPoint(pt1_2,m_Matrix);
        edge.Pt1_3D=pt1_1;
        edge.Pt2_3D=pt1_2;
        m_Body.EdgePlane[1].loop_out.InsertAt(i,edge);
        m_Body.EdgePlane[1].loop_out.RemoveAt(i+1);
    }
    for(i=0;i<m_Body.EdgePlane[0].in_num;i++){    //内环
        for(int k=0;k<m_Body.EdgePlane[0].loop_in[i].GetSize();k++){
            for(int m=0;m<2;m++){
                pt0_1=m_Body.EdgePlane[m].loop_in[i].GetAt(k).Pt1_3D;
                pt0_2=m_Body.EdgePlane[m].loop_in[i].GetAt(k).Pt2_3D;
                GetNewPoint(pt0_1,m_Matrix);
                GetNewPoint(pt0_2,m_Matrix);
                edge.Pt1_3D=pt0_1;
                edge.Pt2_3D=pt0_2;
                m_Body.EdgePlane[m].loop_in[i].InsertAt(k,edge);
                m_Body.EdgePlane[m].loop_in[i].RemoveAt(k+1);
            }
        }
    }
}
```

其中,三维图形顶点的矩阵变换代码参考如下:

```
/*****************************************************************
    GetNewPoint: 三维图形顶点的几何变换
    m_point: 几何变换的三维图形顶点; m_Matrix: 几何变换矩阵
 *****************************************************************/
void GetNewPoint(CPoint3D& m_point,double m_Matrix[][4]){
    CPoint3D point_new;
    double s_dbl;
    point_new.x=m_point.x*m_Matrix[0][0]+m_point.y*m_Matrix[1][0]+m_point.z*m_
Matrix[2][0]+point_new.s*m_Matrix[3][0];
    point_new.y=m_point.x*m_Matrix[0][1]+m_point.y*m_Matrix[1][1]+m_point.z*m_
Matrix[2][1]+point_new.s*m_Matrix[3][1];
    point_new.z=m_point.x*m_Matrix[0][2]+m_point.y*m_Matrix[1][2]+m_point.z*m_
```

```
Matrix[2][2] + point_new.s * m_Matrix[3][2];
s_dbl = m_point.x * m_Matrix[0][3] + m_point.y * m_Matrix[1][3] + m_point.z * m_Matrix[2][3] +
point_new.s * m_Matrix[3][3];
        point_new.x/ = s_dbl;point_new.y/ = s_dbl;point_new.z/ = s_dbl;
        m_point = point_new;
    }
```

5.3.3 投影变换

把三维物体变为二维图形表示的过程称为投影变换，它是三维空间形体在屏幕、打印机、绘图仪等输出设备上显示时必不可少的操作步骤。投影变换相当于将三维形体压缩到一个二维投影平面上表示，因此，投影变换实现的是物体向投影面的投影。在“机械制图”课程中，将三维形体画在平面图纸上就是利用投影变换实现的。根据投影中心和投影面的距离关系，投影分两大类。

(1) 透视投影——投影中心和投影面之间的距离是有限的，如图 5.3-5(a)所示。该投影方法又称中心投影法。

(2) 平行投影——投影中心到投影面的距离是无限的。平行投影又分正投影和斜投影两种：当投影方向即投射线和投影面是垂直关系时，称为正投影，如图 5.3-5(b)所示；当投影方向和投影面非垂直而呈一个倾斜角度时，称为斜投影，如图 5.3-5(c)所示。一般情况下，我们所称的投影和投影变换指的是平行投影中的正投影变换。

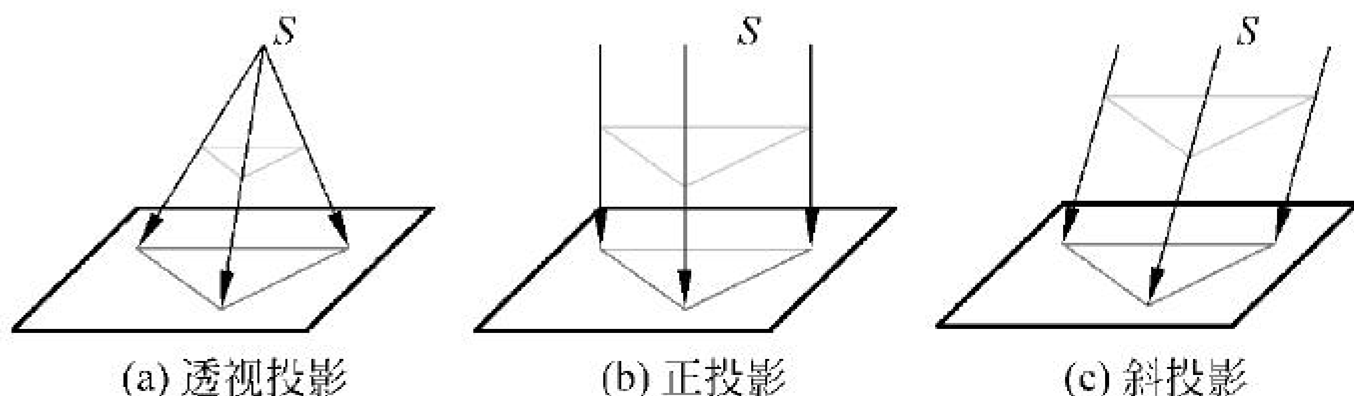


图 5.3-5　投影变换的类型

不同的投影类型对应不同的投影变换方法。对于三维线框体在显示器屏幕上的投影显示，即是将三维空间形体上的顶点投影到显示器屏幕上，然后，将屏幕上相关顶点的投影连线，即得三维形体上对应棱边的投影，当实现空间形体上所有的顶点、棱边在显示屏幕上的投影后，即获得整个三维线框体的投影。

由于显示器屏幕所在平面为 xOy 平面，形体向 xOy 平面投影，相当于将图形压缩到 xOy 平面显示，对应的投影变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}$$

则三维空间点的投影变换为

$$[x^* \quad y^* \quad z^* \quad 1]=[x \quad y \quad z \quad 1]\boldsymbol{T}=[x \quad y \quad 0 \quad 1]$$

即三维空间点在 xOy 平面投影点的坐标是该点空间坐标的 x 和 y 坐标。因此，空间形体向

屏幕投影时,只取顶点的 x 和 y 坐标,即得投影点的坐标。

上述三维线框拉伸体在显示器屏幕上的投影显示函数代码参考如下:

```
/**********************************************************************
    DrawBody: 三维拉伸线框体在显示器屏幕上的投影
pDC: 显示器指针; m_Body: 拉伸线框体; m_DrawColor: 颜色; lineWidth: 线宽; lineType: 线型
    **********************************************************************/
    void DrawBody(CDC * pDC,CBody_Stretch &m_Body,COLORREF &m_DrawColor,int lineWidth = 0,int
lineType = 0){//将拉伸线框体投影到 xOy 平面并连线
        CPoint pt0_1,pt1_1,pt0_2,pt1_2;
        for(int i = 0;i < m_Body.EdgePlane[0].loop_out.GetSize();i++){//上下外环连线
            pt0_1 = m_Body.EdgePlane[0].loop_out.GetAt(i).Pt1_3D.To2DPt();
            pt0_2 = m_Body.EdgePlane[0].loop_out.GetAt(i).Pt2_3D.To2DPt();
            pt1_1 = m_Body.EdgePlane[1].loop_out.GetAt(i).Pt1_3D.To2DPt();
            pt1_2 = m_Body.EdgePlane[1].loop_out.GetAt(i).Pt2_3D.To2DPt();
            MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);          //连线
            MIDPOINT_Line(pDC,pt0_1,pt1_1,m_DrawColor,lineWidth);          //连线
            MIDPOINT_Line(pDC,pt1_1,pt1_2,m_DrawColor,lineWidth);          //连线
        }
        for(i = 0;i < m_Body.EdgePlane[0].in_num;i++){//上下内环连线
            for(int k = 0;k < m_Body.EdgePlane[0].loop_in[i].GetSize();k++){
                pt0_1 = m_Body.EdgePlane[0].loop_in[i].GetAt(k).Pt1_3D.To2DPt();
                pt0_2 = m_Body.EdgePlane[0].loop_in[i].GetAt(k).Pt2_3D.To2DPt();
                pt1_1 = m_Body.EdgePlane[1].loop_in[i].GetAt(k).Pt1_3D.To2DPt();
                pt1_2 = m_Body.EdgePlane[1].loop_in[i].GetAt(k).Pt2_3D.To2DPt();
                MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);      //连线
                MIDPOINT_Line(pDC,pt0_1,pt1_1,m_DrawColor,lineWidth);      //连线
                MIDPOINT_Line(pDC,pt1_1,pt1_2,m_DrawColor,lineWidth);      //连线
            }
        }
    }
```

其中,To2DPt()是在空间顶点类中定义的将空间点转化为屏幕点的变量函数。

在应用程序中创建拉伸线框体时,首先在视图类中增加三维线框体数组变量、实体数目变量及其他相关变量和函数:

```
CBody_Stretch m_Body[100];                              //三维线框体数组变量
int num_Body;                                           //三维线框体数量
CBody_Stretch m_Body_Tmp;                               //用于操作的临时实体变量
C3DStretchDlg * m_3DStretchDlg;                         //三维线框体拉伸对话框
void CreateTmpStretchBody(double& m_dblLength);         //创建临时实体
void CreateBodyforStretch(double& m_dblLength);         //确定创建实体
```

当拾取了用于创建拉伸体的平面多边形后,选择实体拉伸操作,弹出拉伸对话框,输入拉伸长度,调用视图类的 CreateTmpStretchBody()函数,创建临时实体,并投影显示。参考代码如下:

```
void CCGTest002View::CreateTmpStretchBody(double& m_dblLength){
    if(this -> m_Picker.picktype == pick_polyline){//从拾取多边形拉伸实体
        if(CreateBodyOfStretch(m_Body_Tmp,m_Picker,m_dblLength) == true){
```

```
//为了显示立体感,绕第一个点沿 x 轴旋转 30°,再沿 y 轴旋转 30°,然后再显示
        double m_Matrix[4][4],m_Matrix0[4][4];
        CPoint3D pt3D;
        pt3D = m_Body.EdgePlane[0].loop_out.GetAt(0).Pt1_3D;
        GetMatrix(m_Matrix,0,pt3D.x * ( - 1),pt3D.y * ( - 1),pt3D.z * ( - 1),0,1);
                                                         //移动到原点
        GetMatrix(m_Matrix0,1,0,0,0, - 30,1);             //沿 x 轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);               //矩阵级联
        GetMatrix(m_Matrix0,2,0,0,0, - 30,1);             //沿 y 轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);               //矩阵级联
        GetMatrix(m_Matrix0,0,pt3D.x,pt3D.y,pt3D.z,0,1); //移回原位置
        MatrixXMatrix(m_Matrix,m_Matrix0);               //矩阵级联
        GetNewPoint(m_Body,m_Matrix);                    //实体乘以变换矩阵,得新点
        Invalidate();
        }
    }
}
```

在 OnDraw()函数中显示临时实体,相关代码参考如下:

```
if(this - > m_Picker.picktype!= pick_none||m_Picker.m_capture_point.GetSize()> 0) {
if(this - > m_Picker.picktype = = pick_polyline&&m_Body_Tmp.polyline.m_PolyLine_array_Out.
GetSize()> 0)                                            //首先判断是否有临时实体
        DrawBody(pDC,m_Body_Tmp,m_DrawColor);            //画实体
    else
        DrawPicker(pDC,m_Picker,HIGHLIGHTCOLOR,1);
}
```

上述代码形成的拉伸线框体效果如图 5.3-6 所示。

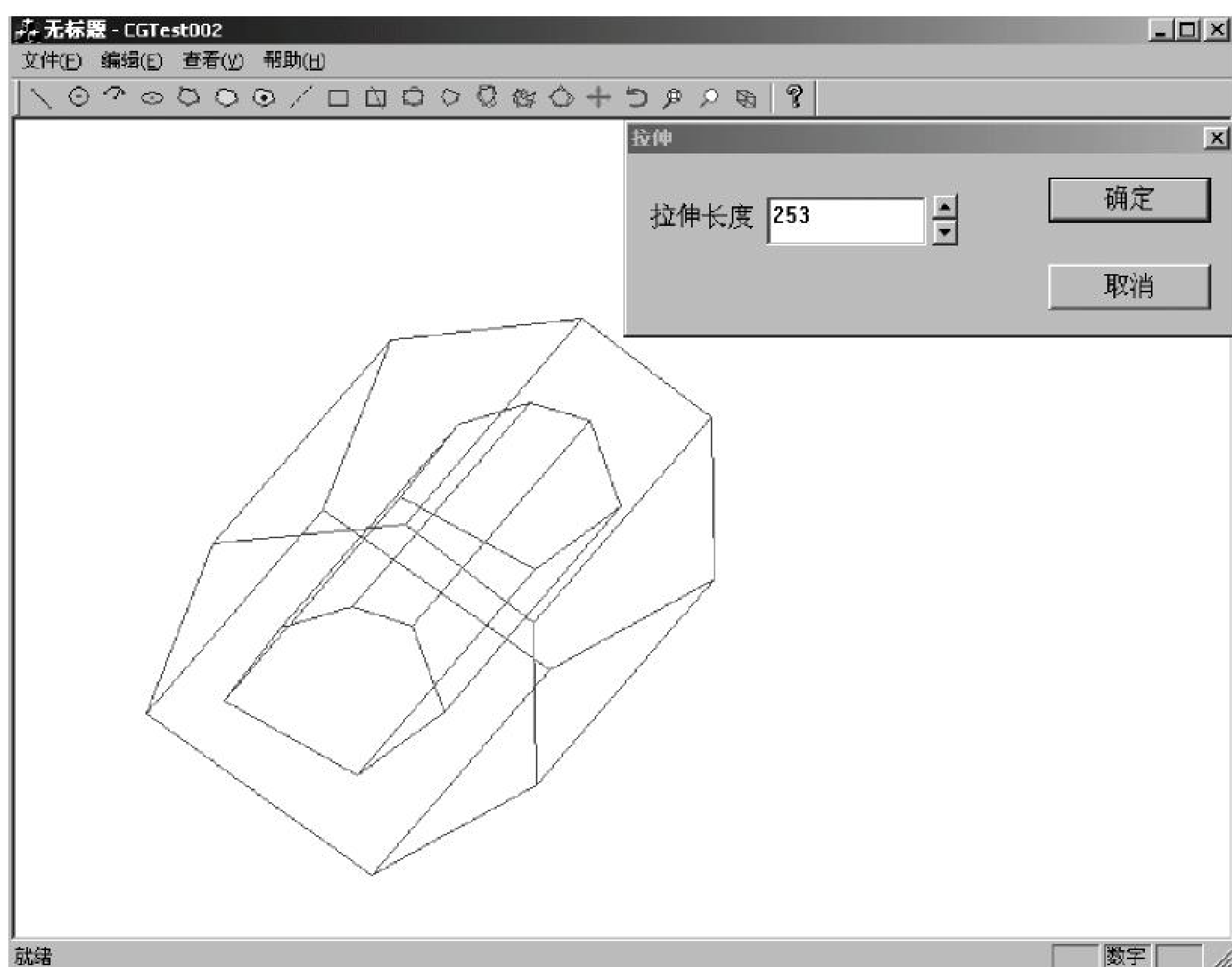


图 5.3-6　拉伸形成线框体

拉伸线框体确定后，除了创建线框体外，形成拉伸体的平面多边形将不再作为独立的图元以供拾取，因此，应在多边形集合中去掉该多边形。代码参考如下：

```
void CCGTest002View:: CreateBodyforStretch(double& m_dblLength){
    if(this->m_Picker.picktype == pick_polyline){
if(CreateBodyOfStretch(m_Body[num_Body++],m_Picker,m_dblLength) == true){
            m_Body_Tmp.ClearBody();          //删除临时实体
    for(int i = 0;i < this->iPolyLine;i++){  //从多边形数组中去掉该拾取的多边形
if(m_Picker.picktype == pick_polyline&&(m_Picker.m_PolyLine == m_PolyLine[i]))
{//判断是否为拾取的图元,如是,则删除这个多边形
m_PolyLine[i].~CPolyLine();
    for(;i < this->iPolyLine - 1;i++)        //循环把后面的多边形赋给前一个
                    m_PolyLine[i] = m_PolyLine[i + 1];
                    break;
                }
            }
            this->iPolyLine--;                //多边形个数减一
            this->m_Picker.picktype = pick_none;
            Invalidate();}
    }
}
```

5.3.4 三维形体的交互技术

和二维图形变换一样，三维图形变换也会用到交互技术。在图元拾取操作中，除了能够拾取二维图形外，对所创建的三维线框拉伸体也应具有拾取功能，为此，需要在拾取结构类中增加线框拉伸体图元。代码参考如下：

```
class CPicker:CDraw{
public:
    //拾取的图元,其他内容前文已列出,不再赘述,仅列出线框拉伸体的图元
    CBody_Stretch m_Body_Stretch;
};
```

同理，在拾取枚举结构中，增加实体拾取类型：

```
enum Picktype { pick_none,pick_line,...,pick_body};
```

三维拉伸体的具体选取步骤是：当光标在拉伸体的表面上时，表示该拉伸体被识别到，则用高亮度颜色或者更粗的线型来显示该实体。和二维图形不同，鼠标在屏幕上拾取的三维形体实际是该形体在屏幕上的投影，拾取到形体的投影即认为该形体被拾取。

由于拉伸实体是通过拉伸形成的，所以，拉伸实体上下两个表面的投影是多边形，侧面投影都是四边形，只要光标在其中一个多边形或者四边形内部，即认为拾取到该拉伸体。一种简单判断光标点是否在多边形内部的方法是扫描线法。

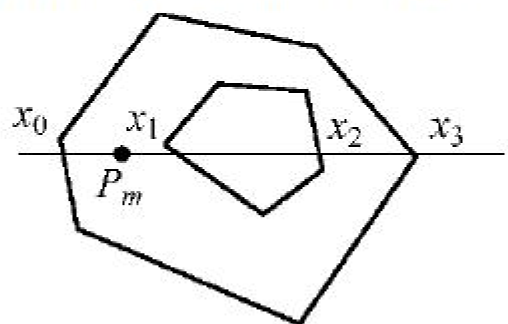


图 5.3-7 判断点在多边形内部

设光标当前点是 $P_m(x_m, y_m)$，令扫描线 $y=y_m$ 与多边形相交，如图 5.3-7 所示。将交点的 x 坐标值按递增

顺序排序,并顺序两两组成区间对,如$[x_0, x_1]$、$[x_2, x_3]$,如果光标点的坐标 x_m 在某一个区间对内,说明光标点在拉伸体的表面上,则拾取该实体。

下面是采用上述方法判断某拉伸实体是否被鼠标拾取的函数代码,可供参考。

```
/*************************************************************************
    CheckIsPicked: 判断某拉伸实体是否被鼠标拾取的函数
point: 鼠标点; Body: 判断的拉伸实体; m_Picker: 图形拾取类
    ********************************************************************/
    bool CheckIsPicked(CPoint &point,CBody_Stretch &Body,CPicker &m_Picker){
        int pFlag = 0;
        /*首先构造上下两个表面的投影多边形(同时得到多边形的最小边界矩形)*/
        CPoint pt0,pt1;
        int xmin = 0,xmax = 0,ymin = 0,ymax = 0;                    //构造最小边界矩形
        CLine line;
        CArray<CLine,CLine> loop;                                   //环
        CPolyLine polyline;
        for(int m = 0;m<2;m++){
        for(int i = 0;i<Body.EdgePlane[m].loop_out.GetSize();i++){  //构造投影多边形外环
            pt0 = Body.EdgePlane[m].loop_out.GetAt(i).Pt1_3D.To2DPt(); //投影点
            pt1 = Body.EdgePlane[m].loop_out.GetAt(i).Pt2_3D.To2DPt(); //投影点
            line.pt1 = pt0;line.pt2 = pt1;loop.Add(line);
            BuildRectEdge(pt0,xmin,xmax,ymin,ymax);                  //构造最小边界矩形
            BuildRectEdge(pt1,xmin,xmax,ymin,ymax);                  //构造最小边界矩形
            }
            polyline.m_PolyLine_array_Out.Append(loop);
            for(i = 0;i<Body.EdgePlane[m].in_num;i++){//构造投影多边形内环
                loop.RemoveAll();
                for(int k = 0;k<Body.EdgePlane[m].loop_in[i].GetSize();k++){
                    pt0 = Body.EdgePlane[m].loop_in[i].GetAt(k).Pt1_3D.To2DPt();
                    pt1 = Body.EdgePlane[m].loop_in[i].GetAt(k).Pt2_3D.To2DPt();
                    line.pt1 = pt0;line.pt2 = pt1;loop.Add(line);}
                polyline.m_PolyLine_array_in[i].Append(loop);}
            polyline.in_num = Body.EdgePlane[m].in_num;              //内环数量
            //判断是否在多边形最小矩形边界内
            if(CheckIsInBox(point,xmin,xmax,ymin,ymax) == false)
                continue;
            //判断光标是否在多边形内部
            if(CheckPtInPolyline(point,polyline) == true){
                pFlag = 1;
                break;
            }
        }
    if(pFlag == 0) { //如不在,继续判断是否在上下边拉伸的侧面多边形投影内
        CPoint pt2,pt3;
        for(int i = 0;i<Body.EdgePlane[0].loop_out.GetSize();i++) {  //外环边
            loop.RemoveAll();
            pt0 = Body.EdgePlane[0].loop_out.GetAt(i).Pt1_3D.To2DPt(); //投影点
            pt1 = Body.EdgePlane[0].loop_out.GetAt(i).Pt2_3D.To2DPt(); //投影点
            line.pt1 = pt0;line.pt2 = pt1;loop.Add(line);
            pt2 = Body.EdgePlane[1].loop_out.GetAt(i).Pt1_3D.To2DPt(); //投影点
```

```
                pt3 = Body.EdgePlane[1].loop_out.GetAt(i).Pt2_3D.To2DPt(); //投影点
                line.pt1 = pt1;line.pt2 = pt3;loop.Add(line);
                line.pt1 = pt3;line.pt2 = pt2;loop.Add(line);
                line.pt1 = pt2;line.pt2 = pt0;loop.Add(line);
                BuildRectEdge(pt2,xmin,xmax,ymin,ymax);              //构造最小边界矩形
                BuildRectEdge(pt3,xmin,xmax,ymin,ymax);              //构造最小边界矩形
                polyline.m_PolyLine_array_Out.Append(loop);
                //判断是否在多边形最小矩形边界内
                if(CheckIsInBox(point,xmin,xmax,ymin,ymax) == false)
                    continue;
                //判断光标是否在多边形内部
                if(CheckPtInPolyline(point,polyline) == true) {
                    pFlag = 1;
                    break;
                    }
                }
                //判断是否在上下内环边拉伸的侧面多边形投影内
                if(pFlag == 0){
                    for(i = 0;i < Body.EdgePlane[0].in_num;i++){           //内环数量
                        for(int k = 0;k < Body.EdgePlane[0].loop_in[i].GetSize();k++){  //内环
                            loop.RemoveAll();
                            pt0 = Body.EdgePlane[0].loop_in[i].GetAt(k).Pt1_3D.To2DPt();
                            pt1 = Body.EdgePlane[0].loop_in[i].GetAt(k).Pt2_3D.To2DPt();
                            line.pt1 = pt0;line.pt2 = pt1;loop.Add(line);
                            pt2 = Body.EdgePlane[1].loop_in[i].GetAt(k).Pt1_3D.To2DPt();
                            pt3 = Body.EdgePlane[1].loop_in[i].GetAt(k).Pt2_3D.To2DPt();
                            line.pt1 = pt1;line.pt2 = pt3;loop.Add(line);
                            line.pt1 = pt3;line.pt2 = pt2;loop.Add(line);
                            line.pt1 = pt2;line.pt2 = pt0;loop.Add(line);
                            polyline.m_PolyLine_array_Out.Append(loop);
                            //判断是否在多边形最小矩形边界内
                            if(CheckIsInBox(point,xmin,xmax,ymin,ymax) == false)
                                continue;
                            //判断光标是否在多边形内部
                            if(CheckPtInPolyline(point,polyline) == true) {
                                pFlag = 1;
                                break;
                            }
                        }
                    }
                }
            }
            if(pFlag == 1){ /* 有拾取,首先判断是否原拾取,如是,则设置状态 = 1 表明已拾取,且在
显示,不必再显示,否则先利用异或方法画原图形,再拾取新图形 */
if(m_Picker.picktype == pick_body&&m_Picker.m_Body_Stretch.polyline == Body.polyline)
m_Picker.stateFlag = 0;
        else {//重新获得拾取的对象
            m_Picker.stateFlag = 1;
            m_Picker.picktype = pick_body;
            m_Picker.m_Body_Stretch = Body;
            }
```

```
        return true;
        }
        return false;
    }
```

上述函数代码中,利用顶点投影构造多边形最小尺寸边界的函数代码为:

```
void BuildRectEdge(CPoint &Point,int &xmin,int &xmax,int &ymin,int &ymax){
    if(Point.x<xmin) xmin=Point.x;
    else if(Point.x>xmax) xmax=Point.x;
    if(Point.y<ymin)ymin=Point.y;
    else if(Point.y>ymax)ymax=Point.y;
}
```

函数中,判断鼠标点是否在投影多边形内部的扫描线判断方法的函数代码为:

```
bool CheckPtInPolyline(CPoint &Point,CPolyLine &polyline){
    CArray<CLine,CLine> m_line_Array_Out;
    m_line_Array_Out.Append(polyline.m_PolyLine_array_Out);          //外环多边形
    CArray<int,int> m_x_Array;                                       //交点 x 坐标集合
    int m_x;                                                         //交点
    int j,k;
    int yi=Point.y;                                                  //扫描线
    m_x_Array.RemoveAll();
    //判断扫描线和哪些边相交,如相交,求交点,并排序
for(int i=0;i<m_line_Array_Out.GetSize();i++){//首先判断扫描线和外环多边形的交点
/*将每条边的最大 y 值缩短一个单位,判断是否和扫描线相交,如相交,求交点,插入交点集并排
序*/
if((yi>=m_line_Array_Out.GetAt(i).pt1.y&&yi<m_line_Array_Out.GetAt(i).pt2.y)||(yi>=m_
line_Array_Out.GetAt(i).pt2.y&&yi<m_line_Array_Out.GetAt(i).pt1.y)){  //求交点
m_x=GetInterPtXForScanY(yi,m_line_Array_Out.GetAt(i).pt1.x,m_line_Array_Out.GetAt(i).
pt1.y,m_line_Array_Out.GetAt(i).pt2.x,m_line_Array_Out.GetAt(i).pt2.y);
            OrderToInsertPt_x(m_x_Array,m_x);                        //排序
        }
        else if(yi==m_line_Array_Out.GetAt(i).pt1.y&&yi==m_line_Array_Out.GetAt(i).pt2.y) {
            //是水平线,则将两个端点加入点集
            OrderToInsertPt_x(m_x_Array,m_line_Array_Out.GetAt(i).pt1.x);
            OrderToInsertPt_x(m_x_Array,m_line_Array_Out.GetAt(i).pt2.x);
        }
    }
    //再判断扫描线和内环多边形的交点
    CArray<CLine,CLine> m_line_Array_in;
    for(k=0;k<polyline.in_num;k++){
        m_line_Array_in.Append(polyline.m_PolyLine_array_in[k]);      //内环多边形
        for(i=0;i<m_line_Array_in.GetSize();i++){
if((yi>=m_line_Array_in.GetAt(i).pt1.y&&yi<m_line_Array_in.GetAt(i).pt2.y)||(yi>=m_
line_Array_in.GetAt(i).pt2.y&&yi<m_line_Array_in.GetAt(i).pt1.y)){ //求交点
m_x=GetInterPtXForScanY(yi,m_line_Array_in.GetAt(i).pt1.x,m_line_Array_in.GetAt(i).pt1.
y,m_line_Array_in.GetAt(i).pt2.x,m_line_Array_in.GetAt(i).pt2.y);
        OrderToInsertPt_x(m_x_Array,m_x);                            //排序
        }
        else if(yi==m_line_Array_in.GetAt(i).pt1.y&&yi==m_line_Array_in.GetAt(i).pt2.
```

```
y){
            //是水平线,则将两个端点加入点集
            OrderToInsertPt_x(m_x_Array,m_line_Array_in.GetAt(i).pt1.x);
            OrderToInsertPt_x(m_x_Array,m_line_Array_in.GetAt(i).pt2.x);
            }
        }
    m_line_Array_in.RemoveAll();}
    //判断是否在区间对内
    for(j=0;j<=m_x_Array.GetSize()-2;j++,j++){
        if(Point.x>=m_x_Array.GetAt(j)&&Point.x<=m_x_Array.GetAt(j+1))
            return true;}
    return false;
}
```

在应用程序的鼠标移动消息函数中,调用判断鼠标点是否拾取实体的函数为:

```
bool CheckIsPicked(CPoint &point,CBody_Stretch *m_Body,int &body_num,CPicker &m_Picker){
//判断是否拾取了某实体
    int pFlag=0;                                                    //是否拾取
    CLine line;
    for(int i=0;i<body_num;i++){
        if(CheckIsPicked(point,m_Body[i],m_Picker)==true)
            return true;
    }
    return false;
}
```

如图5.3-8所示,当鼠标在一个拉伸实体的表面时,该实体以高亮度颜色和粗线显示。

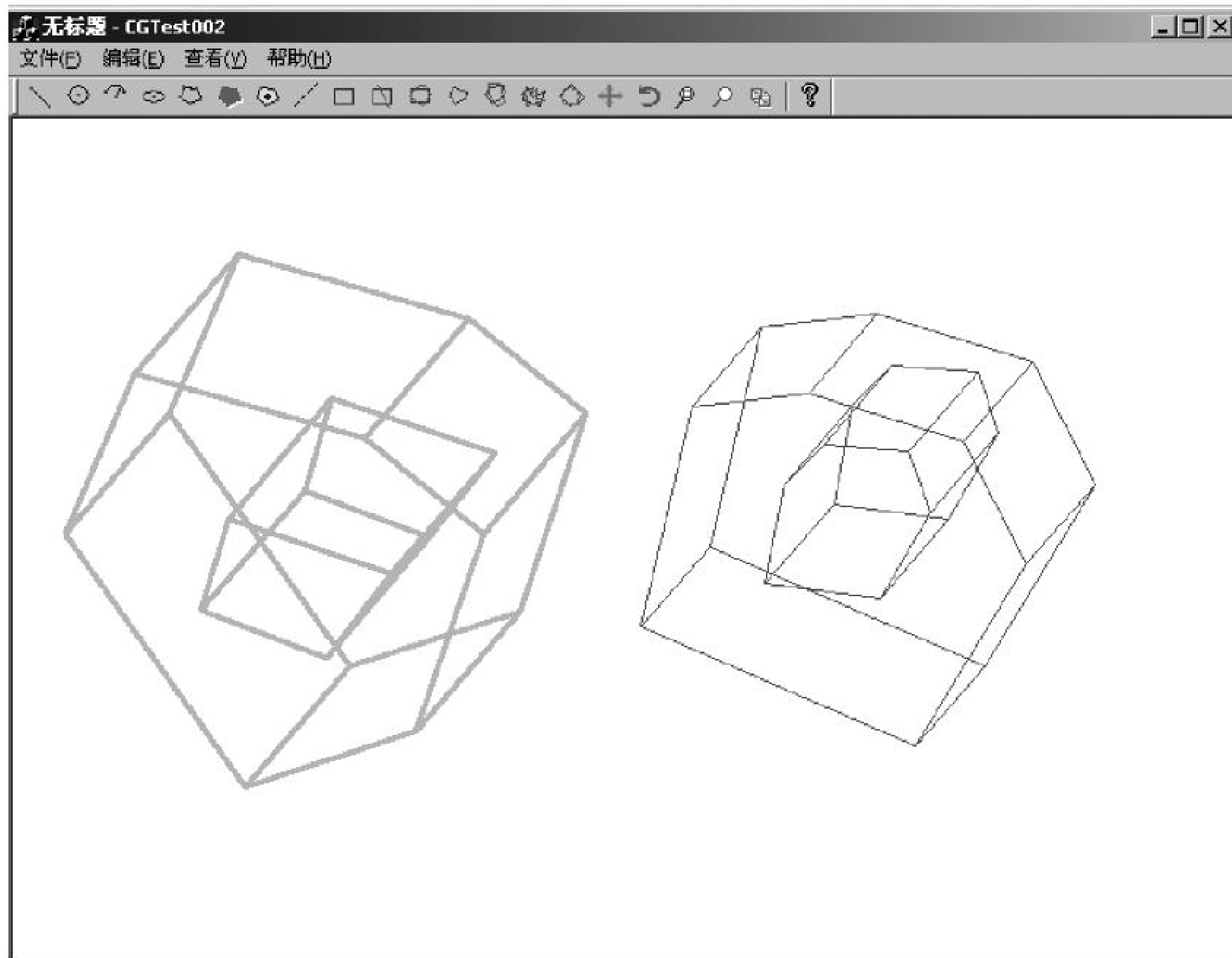


图5.3-8　实体拾取

在视图类的 OnDraw()中显示实体时，对于拾取的实体，在拾取图形的绘图函数 DrawPicker()中绘制，因此，需要在 DrawPicker()中增加绘制实体的选项。DrawPicker()中增加判断是否绘制实体的代码如下：

```
if(m_Picker.picktype == pick_body)                                    //绘制拾取的实体
    DrawBody(pDC,m_Picker.m_Body_Stretch,crColor,lineWidth);         //画拾取实体
```

对于未拾取的实体，则直接在 OnDraw()中绘制。OnDraw()增加绘制实体的代码为：

```
for(int i = 0;i < num_Body;i++) { //画实体
if((m_Picker.picktype == pick_body&&m_Body[i] == m_Picker.m_Body_Stretch) == false)
            DrawBody(pDC,m_Body[i],m_DrawColor);
    }
```

对于拾取的实体即可实现各种图形变换，如平移、旋转、缩放等。各种图形变换可借助鼠标或者对话框来实现。例如，通过鼠标移动实体，则在鼠标移动消息函数中，增加移动变换处理代码，参考如下：

```
if(this -> m_iFlag == 8){                                        //m_iFlag = 8,表示移动变换
    if(nFlags == MK_LBUTTON&&pickstep == 1) {                    //是否左键被按下
        pickPt2 = point;
        int x_dis = pickPt2.x - pickPt1.x;
        int y_dis = pickPt2.y - pickPt1.y;
        if(m_Picker.picktype!= pick_body){
            double m_Matrix[3][3];
            Get2DMatrix(m_Matrix,0,0,x_dis,y_dis,0);             //二维移动变换矩阵
            TransforOf2D_Single(m_Matrix);                        //二维变换
            pickPt1 = point;
            Invalidate();
        }
        else {
            double m_Matrix[4][4];
            GetMatrix(m_Matrix,0,x_dis,y_dis,0,0,0);             //三维移动变换矩阵
            GetNewPoint(m_Picker.m_Body_Stretch,m_Matrix);        //三维变换
            pickPt1 = point;
            Invalidate();
        }
    }
```

三维形体的旋转变换，可以利用非模式对话框选择旋转轴和输入旋转角度，如图 5.3-9 所示。在对话框中，修改旋转角度，调用视图类中的旋转变换函数 Rotate3D()，可以实时获得旋转效果。函数代码参考如下：

```
void CCGTest002View::Rotate3D(int &m_iAxis,double &angle){
    if(m_Picker.picktype == pick_body){
        for(int i = 0;i < num_Body;i++){
            if(m_Body[i] == m_Picker.m_Body_Stretch){
                m_Picker.m_Body_Stretch = m_Body[i];
                break;}                                  //每次旋转前,拾取的图形重新返回到最
                                                         //初拾取状态
```

```
        }
    }
    double m_Matrix[4][4],m_Matrix0[4][4];
    CPoint3D pt3D;
    //以实体的第一个顶点为旋转中心
    pt3D = m_Picker.m_Body_Stretch.EdgePlane[0].loop_out.GetAt(0).Pt1_3D;
    GetMatrix(m_Matrix,0,pt3D.x * ( - 1),pt3D.y * ( - 1),pt3D.z * ( - 1),0,1);  //移动到原点
    GetMatrix(m_Matrix0,m_iAxis,0,0,0,angle,1);      //m_iAxis 为旋转轴,1 为 x 轴
    MatrixXMatrix(m_Matrix,m_Matrix0);               //矩阵级联
    GetMatrix(m_Matrix0,0,pt3D.x,pt3D.y,pt3D.z,0,1); //移回原位置
    MatrixXMatrix(m_Matrix,m_Matrix0);               //矩阵级联
    GetNewPoint(m_Picker.m_Body_Stretch,m_Matrix);   //实体乘以变换矩阵,得新点
    Invalidate();
}
```

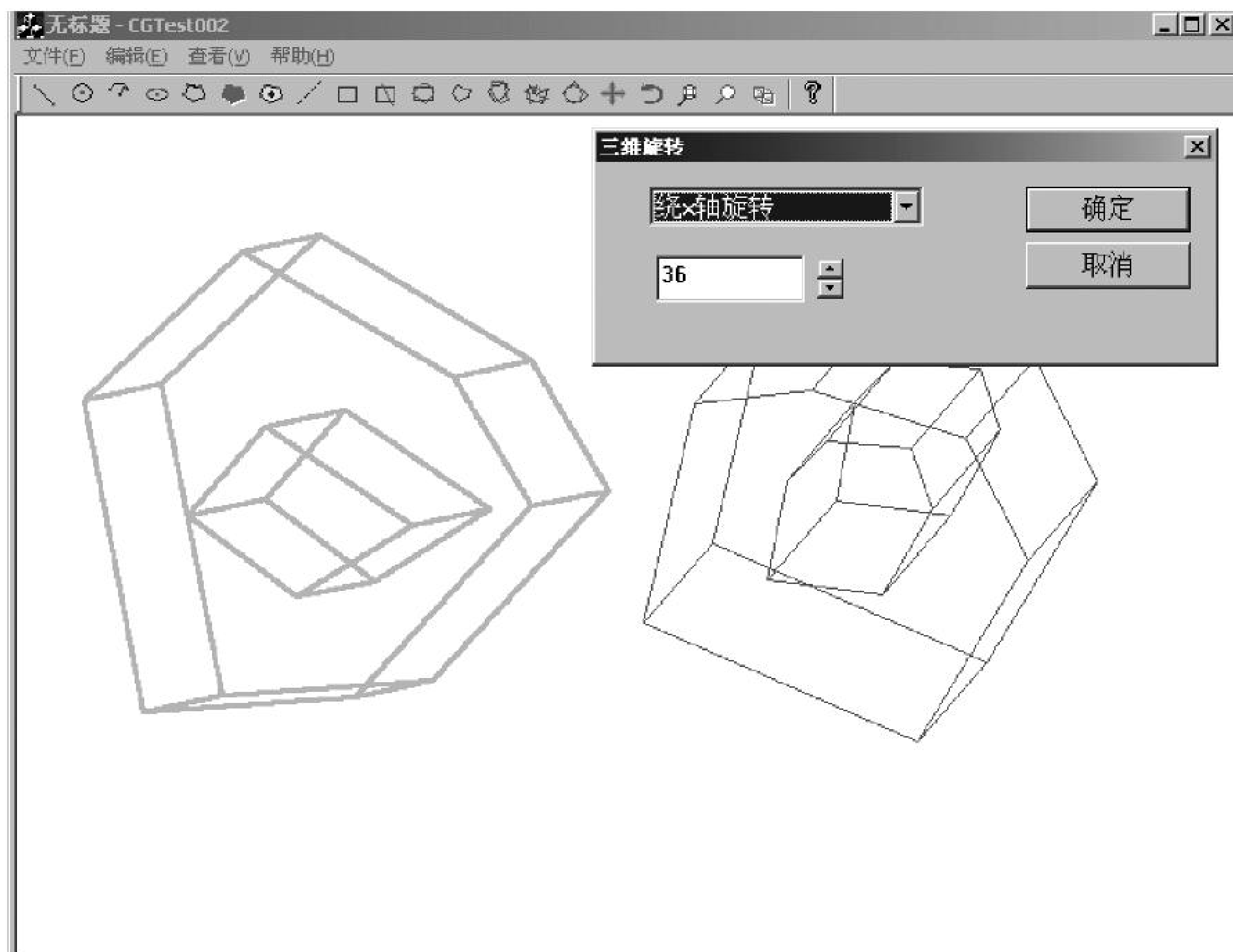


图 5.3-9　旋转变换

当旋转确定后,将旋转后的拾取形体赋给实体集合中的对应实体,相关代码如下:

```
if(m_Picker.picktype == pick_body) {
    for(int i = 0;i < num_Body;i++) {
        if(m_Body[i] == m_Picker.m_Body_Stretch){
            m_Body[i] = m_Picker.m_Body_Stretch;
            break;
        }
    }
}
```

对于"机械制图"课程中使用的三视图、轴测图以及其他基本视图等各种投影图,都可以

推导出对应的投影变换矩阵。

主视图——把立体向 xOz 面(V 面)做正投影,得到主视图,根据正投影原理(平行投影面的投影保持原形),则

$$\begin{cases} x^* = x \\ y = 0 \\ z^* = z \end{cases}$$

主视图投影变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

将投影图形分别沿 x 向和 z 向平移 l_V 和 n_V,则主视图的组合变换矩阵为

$$\boldsymbol{T}_V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_V & 0 & n_V & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ l_V & 0 & n_V & 1 \end{bmatrix}$$

俯视图——把立体向 xOy 面(H 面)做正投影,再将投影(xOy 面)绕 x 轴旋转 $-90°$得到俯视图,也可看成先将立体绕 x 轴旋转 $-90°$再向 xOz 面做正投影,因此

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_H & 0 & n_H & 1 \end{bmatrix}$$

其中,$\theta = -90°$,$l_H = l_V$,俯视图的组合变换矩阵为

$$\boldsymbol{T}_H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ l_V & 0 & n_H & 1 \end{bmatrix}$$

左视图——可以看成先将立体绕 z 轴旋转 $90°$再向 xOz 面做正投影,则

$$\boldsymbol{T} = \begin{bmatrix} \cos\psi & \sin\psi & 0 & 0 \\ -\sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_W & 0 & n_W & 1 \end{bmatrix}$$

其中,$\psi = 90°$,$n_W = n_V$,左视图的组合变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_W & 0 & n_V & 1 \end{bmatrix}$$

轴测投影有正轴测投影和斜轴测投影两种类型。正轴测投影图是正投影,投影面是 V 面,使物体的三个面都与 V 面倾斜——即旋转,先绕 z 轴旋转 ψ 角,再绕 x 轴旋转 θ 角,然后

向 V 面投影。正轴测投影的变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}\cos\psi & \sin\psi & 0 & 0\\ -\sin\psi & \cos\psi & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta & \sin\theta & 0 & 0\\ -\sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ l & 0 & n & 1\end{bmatrix}$$

$$=\begin{bmatrix}\cos\psi & 0 & \sin\psi\sin\theta & 0\\ -\sin\psi & 0 & \cos\psi\sin\theta & 0\\ 0 & 0 & \cos\theta & 0\\ l & 0 & n & 1\end{bmatrix}$$

正等轴测图的三个轴向变形系数相等，可推出 $\psi=45°$，$\theta=-35°16'$。

在“机械制图”中，斜二等轴测图的形成原理是将物体正放，采用平行投影法的斜投影实现。如果物体放正并能看到三个面，可先将物体变形，利用前面讲过的错移变换实现：

要看到左右两侧面——沿 x 含 y 错移；

要看到上下两面——沿 z 含 y 错移。

因此，形成斜轴测图需要三个步骤：①将物体沿 x 含 y 错移；②将物体沿 z 含 y 错移；③将物体向 V 面正投影。斜轴测投影的变换矩阵为

$$\boldsymbol{T}=\begin{bmatrix}1 & 0 & 0 & 0\\ d & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & f & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ l & 0 & n & 1\end{bmatrix}=\begin{bmatrix}1 & 0 & 0 & 0\\ d & 0 & f & 0\\ 0 & 0 & 1 & 0\\ l & 0 & n & 1\end{bmatrix}$$

当 d、f 取不同的值时，可以得到立体各种不同的斜轴测图。由于 d、f 的正负决定错移方向，因此 d、f 的符号可决定斜轴的方向。

对于常用的斜二等轴测投影，根据条件：x 和 z 向变形系数都是 1，y 向变形系数为 0.5，y 轴和水平方向的夹角为 45°，可推出 $d=f=\pm0.354$。f、d 可正可负，当看到物体上表面时取 $f=-0.354$。

5.3.5 透视投影变换

透视投影是当投影中心(即观察视点)到投影面的距离有限时，物体在投影面的投影，距离观察视点近的物体投影大，距离观察点远的物体投影小。透视投影符合人的视觉系统观察物体产生的远近空间层次感，所以在真实感图形中广泛使用。

在透视投影中，视线(投影线)是从观察视点(即投影中心)出发，因此，投影线是不平行的，这样，空间原来平行的棱线透视投影后可能不再平行。若不平行，则其延长线会汇聚为一点，此点称为灭点。只在一个坐标轴方向有灭点的透视图为一点透视，如图 5.3-10(a)所示；在两个坐标轴方向有灭点的透视称为二点透视，如图 5.3-10(b)所示；在三个坐标轴方向有灭点的透视称为三点透视，如图 5.3-10(c)所示。

透视投影变换可通过如下方式进行推导：假设计算空间一点 $P(x,y,z)$的投影，观察视点在 y 轴上的点 $E(0,y_e,0)$，其中 y_e 为观察点到投影面 xOz 的距离，如图 5.3-11 所示。投影后点的坐标与投影前点的坐标的关系可推导为

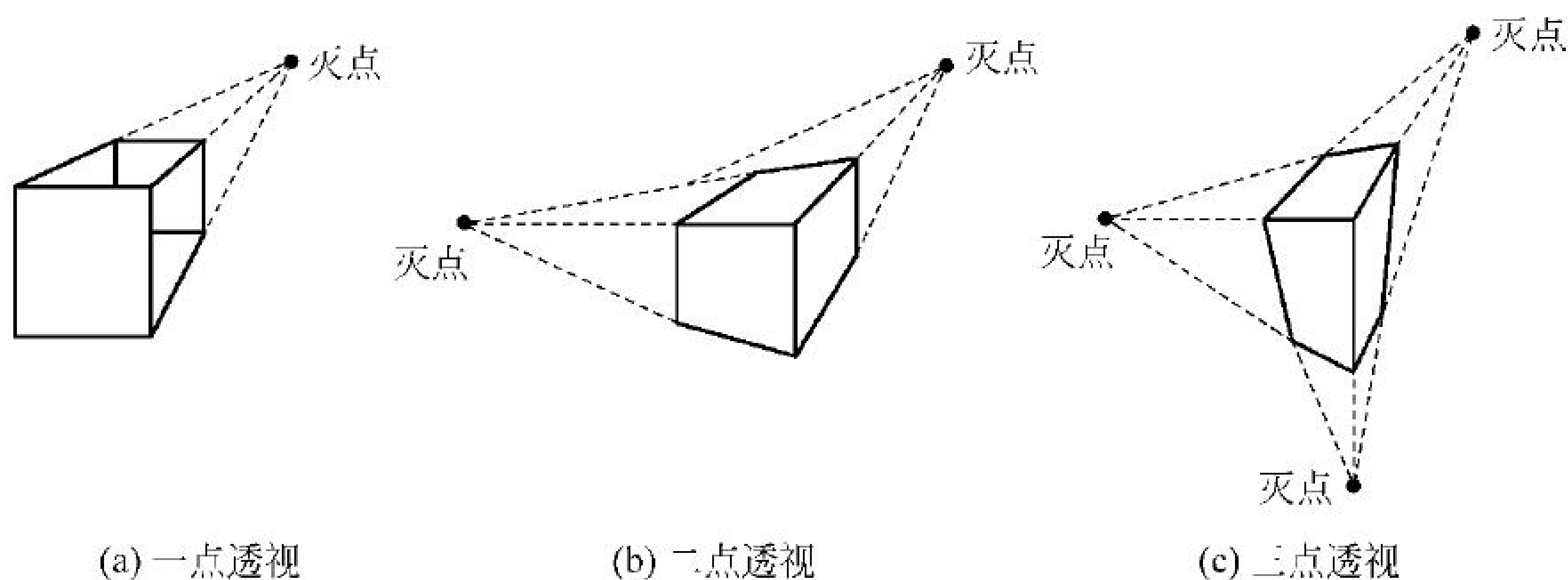


图 5.3-10 透视投影

$$\begin{cases} x^* = -\dfrac{y_e}{y-y_e}x \\ z^* = -\dfrac{y_e}{y-y_e}z \end{cases}$$

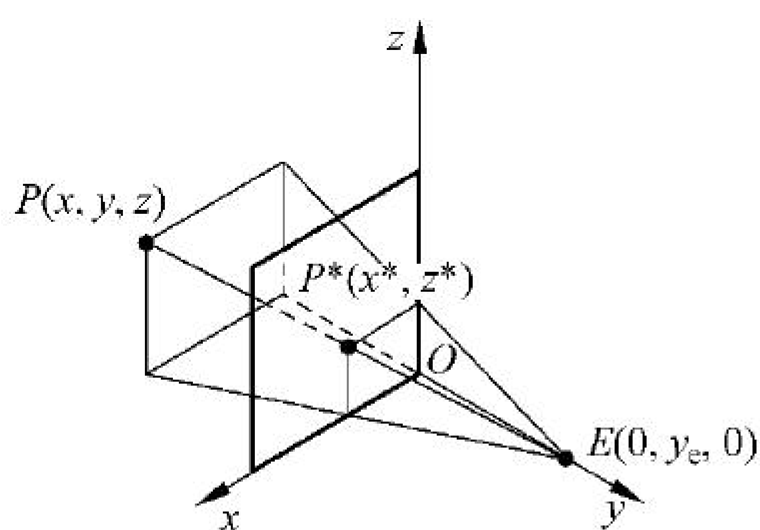


图 5.3-11 简单透视投影变换

设 $q=-\dfrac{1}{y_e}$，代入上式整理可得

$$\begin{cases} x^* = \dfrac{x}{1+qy} \\ z^* = \dfrac{z}{1+qy} \end{cases}$$

用矩阵表示为

$$[x^* \quad y^* \quad z^* \quad 1] = [x \quad y \quad z \quad 1]\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & q\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

$$= [x \quad 0 \quad z \quad 1+qy]$$

规格化后，即为 $\left[\dfrac{x}{1+qy} \quad 0 \quad \dfrac{z}{1+qy} \quad 1\right]$。

从上述公式可以看出，当空间点 $P(x,y,z)$ 距离投影面无穷远时，即 $1+qy\to\infty$，则点 $P(x,y,z)$ 的投影成为 y 轴上的一个点，因为得到的是一个灭点，所以这是一点透视变换。同理，也可推导出观察点在 x 轴上以 yOz 为投影面的一点透视变换矩阵以及观察点在 z 轴上以 xOy 为投影面的一点透视变换矩阵分别为

$$\boldsymbol{T}_{x透视} = \begin{bmatrix}1 & 0 & 0 & p\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}, \quad \boldsymbol{T}_{z透视} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & r\\0 & 0 & 0 & 1\end{bmatrix}$$

从一点透视变换矩阵可以看出，对透视变换起作用的是矩阵第四列的前三行元素 p、q 和 r，一个元素决定一个坐标轴方向的透视投影，那么它们两两组合，就会在两个坐标轴方向形成透视投影，得到两个灭点，即二点透视，三个元素组合在一起就形成三点透视。

在 x 轴和 y 轴方向上的二点透视变换矩阵为

$$T_{xy透视} = \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

在 y 轴和 z 轴方向上的二点透视变换矩阵为

$$T_{yz透视} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

在 x 轴和 z 轴方向上的二点透视变换矩阵为

$$T_{xz透视} = \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

同理，在 x、y 和 z 三个轴方向上的三点透视变换矩阵为

$$T_{xyz透视} = \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

需要注意的是，透视变换和前述的投影变换一样，是空间形体在显示器屏幕等投影面上显示的效果，空间形体的形状并没有改变，形状本身也不存在灭点，因此，只在空间形体向投影面投影时才进行透视变换，获得投影。为此，在应用程序中，需要修改 5.3.3 节中三维拉伸线框体向显示器屏幕上投影绘制函数 DrawBody()，使之既支持平行正投影也支持透视投影。代码参考如下：

```
void DrawBody(CDC * pDC,CBody_Stretch &m_Body,COLORREF &m_DrawColor,double m_Matrix_V[][4],
int lineWidth = 0,int lineType = 0){/ * 参数中加入透视投影变换矩阵 m_Matrix_V * /
    CPoint pt0_1,pt1_1,pt0_2,pt1_2;
    CPoint3D pt_3D;
    for(int i = 0;i < m_Body.EdgePlane[0].loop_out.GetSize();i++){//处理外环
        pt_3D = m_Body.EdgePlane[0].loop_out.GetAt(i).Pt1_3D;
        GetNewPoint(pt_3D,m_Matrix_V);                            //空间点进行透视投影变换
        pt0_1 = pt_3D.To2DPt();
        pt_3D = m_Body.EdgePlane[0].loop_out.GetAt(i).Pt2_3D;
        GetNewPoint(pt_3D,m_Matrix_V);                            //空间点进行透视投影变换
        pt0_2 = pt_3D.To2DPt();
        pt_3D = m_Body.EdgePlane[1].loop_out.GetAt(i).Pt1_3D;
        GetNewPoint(pt_3D,m_Matrix_V);                            //空间点进行透视投影变换
        pt1_1 = pt_3D.To2DPt();
        pt_3D = m_Body.EdgePlane[1].loop_out.GetAt(i).Pt2_3D;
        GetNewPoint(pt_3D,m_Matrix_V);                            //空间点进行透视投影变换
        pt1_2 = pt_3D.To2DPt();
        MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);  //画线
        MIDPOINT_Line(pDC,pt0_1,pt1_1,m_DrawColor,lineWidth);  //画线
```

```
            MIDPOINT_Line(pDC,pt1_1,pt1_2,m_DrawColor,lineWidth);  //画线
        }
        for(i = 0;i < m_Body.EdgePlane[0].in_num;i++){              //处理内环
            for(int k = 0;k < m_Body.EdgePlane[0].loop_in[i].GetSize();k++) {
                pt_3D = m_Body.EdgePlane[0].loop_in[i].GetAt(k).Pt1_3D;
                GetNewPoint(pt_3D,m_Matrix_V);
                pt0_1 = pt_3D.To2DPt();
                pt_3D = m_Body.EdgePlane[0].loop_in[i].GetAt(k).Pt2_3D;
                GetNewPoint(pt_3D,m_Matrix_V);
                pt0_2 = pt_3D.To2DPt();
                pt_3D = m_Body.EdgePlane[1].loop_in[i].GetAt(k).Pt1_3D;
                GetNewPoint(pt_3D,m_Matrix_V);
                pt1_1 = pt_3D.To2DPt();
                pt_3D = m_Body.EdgePlane[1].loop_in[i].GetAt(k).Pt2_3D;
                GetNewPoint(pt_3D,m_Matrix_V);
                pt1_2 = pt_3D.To2DPt();
                MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);   //画线
                MIDPOINT_Line(pDC,pt0_1,pt1_1,m_DrawColor,lineWidth);   //画线
                MIDPOINT_Line(pDC,pt1_1,pt1_2,m_DrawColor,lineWidth);   //画线
            }
        }
}
```

同样,在绘制拾取图形的函数 DrawPicker()中,也需要加入对拾取实体绘制透视投影的变换矩阵,代码参考如下:

```
void DrawPicker(CDC * pDC,CPicker &m_Picker,COLORREF crColor,double m_Matrix_V[][4],int
lineWidth = 0){//m_Matrix_V 为透视投影变换矩阵
    …//此处代码前文已述,省略
    else if(m_Picker.picktype == pick_body){      //判断是否在实体上,画拾取实体
DrawBody(pDC,m_Picker.m_Body_Stretch,crColor,m_Matrix_V,lineWidth);}
    …//此处代码前文已述,省略
}
//m_Matrix_V 为支持透视投影的变换矩阵,在视图类中定义
double m_Matrix_V[4][4];
```

并在视图类的构造函数中,将其初始化为单位矩阵:

```
for(int ii = 0;ii < 4;ii++){
    for(int j = 0;j < 4;j++)
        (m_Matrix_V[ii][j]) = 0;
}
m_Matrix_V[0][0] = 1;
m_Matrix_V[1][1] = 1;
m_Matrix_V[2][2] = 1;
m_Matrix_V[3][3] = 1;
```

为了实现透视变换功能,在应用程序中,创建一个设置透视变换的非模式对话框(例如名称为 CPerspectiveDlg),在对话框中,设置三个坐标轴方向的视点到投影面的距离,当视点无穷远时,即为平行变换,视点和观察点为有限距离时为透视变换。设置一个方向的视点距离为一点透视,设置两个方向及三个方向的视点距离,即为二点透视和三点透视。为了操

作直观，采用滑动控件来设置距离，操作效果如图 5.3-12 所示。

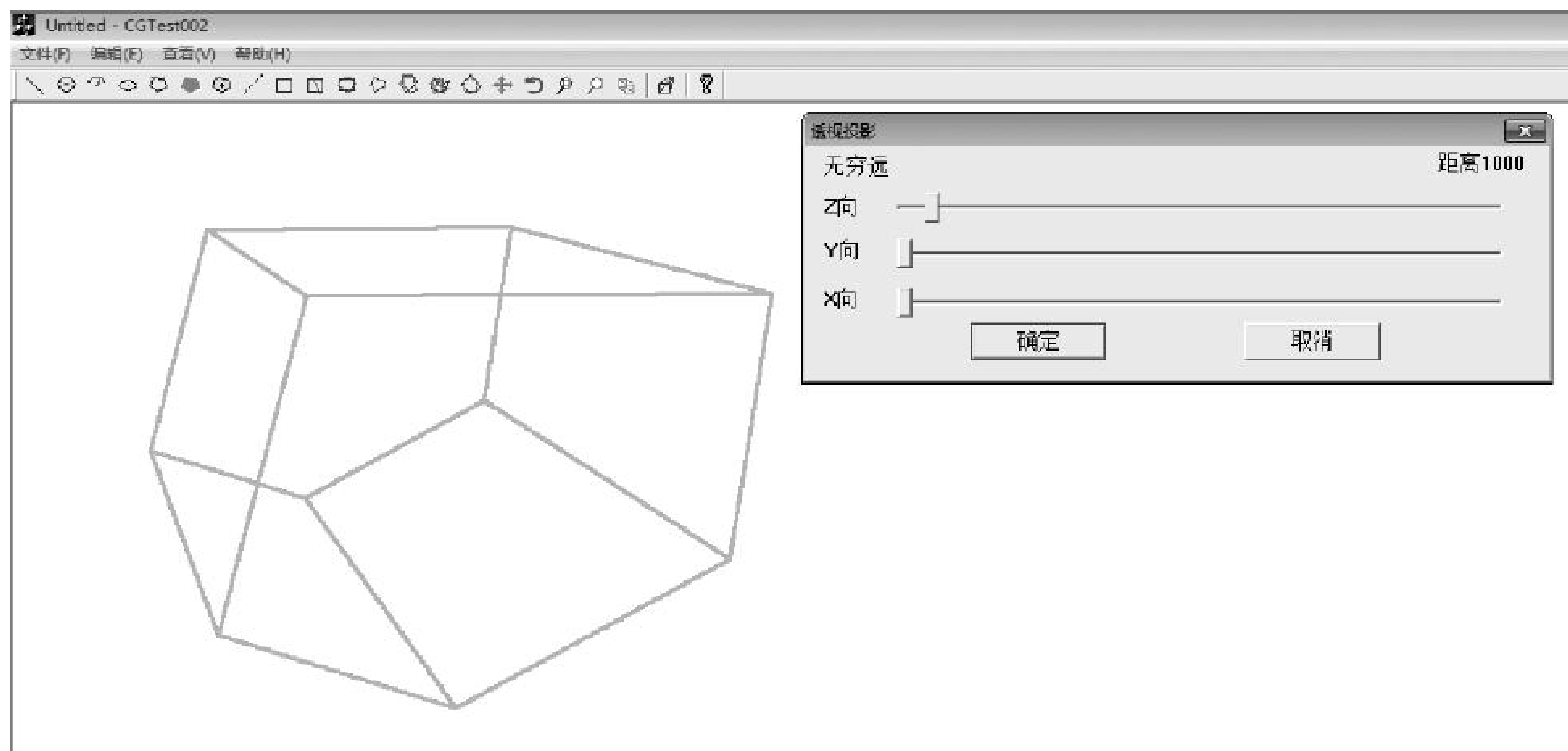


图 5.3-12　透视变换

在设置视点距离时，视点距离太远则透视效果不明显，但是如果太近，由于物体相对投影面的位置，投影会有大的变形。为了具有好的观察效果，视点到投影面的距离大于物体的大小比较合适，例如，形体拉伸 500m 左右，视点到投影面的距离为 1000m。

滑动控件在对话框的初始化函数 OnInitDialog()中设置滑动范围和滑动步长：

```
BOOL CPerspectiveDlg::OnInitDialog() {
    CDialog::OnInitDialog();
    m_x_slider.SetRange(0,1000);                //设置 x 方向滑动控件的滑动范围
    m_y_slider.SetRange(0,1000);                //设置 y 方向滑动控件的滑动范围
    m_z_slider.SetRange(0,1000);                //设置 z 方向滑动控件的滑动范围
    m_x_slider.SetLineSize(50);                 //设置 x 方向滑动控件的滑动步长
    m_y_slider.SetLineSize(50);                 //设置 y 方向滑动控件的滑动步长
    m_z_slider.SetLineSize(50);                 //设置 z 方向滑动控件的滑动步长
    return TRUE;
}
```

在对话框的 OnHScroll()消息函数中获得滑动控件中滑动块的位置，取得视点距离，并设置对应透视投影变换矩阵的元素值：

```
void CPerspectiveDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar) {
    if(pScrollBar->GetDlgCtrlID() == IDC_SLIDER_Z){      //z 方向透视变换
        m_Z = ((CSliderCtrl *)pScrollBar)->GetPos();     //取得滑块位置
        if(m_Z<1000&&m_Z>0)
            m_d = -1.0/(1000 - m_Z);
        else if(m_Z == 1000)
            m_d = -0.01;                        //视点距离太近,为避免失真,给定一个值
        else
            m_d = 0;                            //设置视点无穷远
        this->m_pView->m_Matrix_V[2][3] = m_d;  //设置矩阵值
    }
    else if(pScrollBar->GetDlgCtrlID() == IDC_SLIDER_Y){   //y 方向透视变换
```

```
        m_Y = ((CSliderCtrl * )pScrollBar) -> GetPos();
        if(m_Y < 1000&&m_Y > 0)
            m_d = - 1.0/(1000 - m_Y);
        else if(m_Z == 1000)
            m_d = - 0.01;
        else
            m_d = 0;
        this -> m_pView -> m_Matrix_V[1][3] = m_d;    //设置矩阵值
    }
    else if(pScrollBar -> GetDlgCtrlID() == IDC_SLIDER_X){      //y 方向透视变换
        m_X = ((CSliderCtrl * )pScrollBar) -> GetPos();
        if(m_X < 1000&&m_X > 0)
            m_d = - 1.0/(1000 - m_X);
        else if(m_Z == 1000)
            m_d = - 0.01;
        else
            m_d = 0;
        this -> m_pView -> m_Matrix_V[0][3] = m_d;    //设置矩阵值
    }
    this -> m_pView -> Invalidate();
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

第 6 章

消隐技术

空间形体的形状通过投影变换等操作可在显示屏幕上绘制出来，但是，此时绘制的图形没有深度信息，这会导致图形所表示的空间形体具有二义性(即一个图形可能会有两种理解)，因为真实世界的空间形体的表面之间在视线方向上会存在遮挡关系，形体表面的线和面存在可见和不可见两种情况，被遮挡的线和面是不可见的，投影后不必绘制，或者与可见线和面在绘制时进行区分，投影变换是将形体的所有线面向屏幕上投影，并不判断遮挡关系，这就造成了形体的二义性。为了消除投影变换产生的二义性，在绘制图形时必须消除形体在视线方向上被遮挡的不可见的线和面，这种处理习惯上称作消除隐藏线和隐藏面，简称为消隐。消隐问题是计算机图形学中的一个较为困难又很引人注意的研究课题，至今为止，已有数十种算法被提出来，发表的有关文章已有数百篇。但是由于物体的形状、大小、相对位置等因素千变万化，它仍吸引人们做出不懈努力去探索更好的算法。

6.1 消隐相关概念及算法类型

无论多么复杂的空间物体，当不透光时，在视线方向上都只能看到形体的一部分，其余部分由于光线被遮挡是不可见的，这些部分称为隐藏部分。隐藏部分有隐线和隐面两种类型。

隐线(hidden line)：又叫隐藏线，是在一定的投影变换(平行投影或透视投影)情况下物体上不可能被观察者看到的边或轮廓线。

隐面(hidden surface)：又称隐藏面，是在一定的投影变换(平行投影或透视投影)情况下物体上不可能被观察者看到的表面。

根据消隐类型的不同，消隐算法分为如下两类。

(1) 隐线算法——用于消除物体上不可见的轮廓线。隐线算法主要用于处理线框图形的消隐问题。隐线算法的基本思路是判断面对线的遮挡关系，反复地进行线线、线面之间的求交运算。算法基本步骤是：通过构造参与消隐判断的面表、边表，循环取出一个边，逐次判别该边与面表中不含该边的其他面的遮挡关系，保留并显示该边的可见部分。

(2) 隐面算法——用于消除物体上不可见的表面。隐面算法主要是针对表面模型和实体模型提出的，它不仅可以绘制物体的可见棱边，还可以填充可见的各个表面内的区域。

根据消隐空间的不同，消隐算法又可以分为如下三种类型。

（1）物体空间算法

算法在描述物体的坐标系空间中进行，通过比较物体和物体之间在空间的相对关系，确定可见与不可见。这类算法是将物体表面上的 k 个多边形中的每一个面与其余的 $k-1$ 个面进行比较，精确地求出物体上每条棱边或每个面的遮挡关系。这需运用多种几何计算以达到尽可能高的精度。这类算法的计算量正比于 k^2。

（2）图像空间算法

图像空间是相对于物体空间而言的，即物体经过进一步转换到了屏幕坐标空间。这类算法对屏幕的每一像素进行判断，以决定物体上哪个多边形在该像素点上是可见的。若屏幕上有 $m\times n$ 个像素点，物体表面上有 k 个多边形，则该类消隐算法的计算量将正比于 mnk。此算法的精度与光栅显示屏幕的分辨率有关。后文所述的 Z-buffer 算法、扫描线算法以及 Warnock 算法均属于该类算法。

（3）物像空间消隐算法

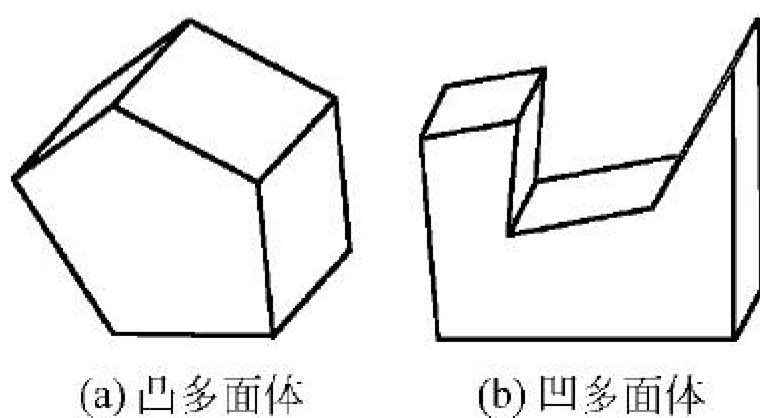
算法同时在物体坐标系空间和图像空间中进行。该类算法首先需要对表面进行优先级排序、深度排序、区域细分或者光线投射等处理，从而判断屏幕的像素点显示哪个表面的颜色。

6.2　凸多面体的消隐

6.2.1　凸、凹多面体的区分

凸多面体是指连接形体中任意两个顶点的线段全部落在该形体中的多面体内。凸多面体是由凸多边形围成的。凸多边形各内角均小于 180°。凸多面体的表面要么完全可见，要么完全不可见，如图 6.2-1(a)所示。在消隐问题中，凸多面体是最简单和最基本的情形，其消隐算法的关键是测试其上哪些表面是可见的，哪些表面是不可见的。

如果顶点间的连线有的落在多面体或多边形之外，则称其为凹多面体，如图 6.2-1(b)所示。凹多面体上有些表面存在着部分可见部分不可见。因此，凹多面体的消隐要比凸多面体的消隐复杂得多。



(a) 凸多面体　　(b) 凹多面体

图 6.2-1　多面体类型

6.2.2　利用平面外法线判断可见性

平面外法线是指垂直相应平面由物体表面指向外部空间的法线，所以称其为外法线。平面的外法线向量 $\boldsymbol{N}$ 可由多边形平面上任意相交且不重合的两线段向量的乘积计算得出。计算外法线向量应从形体外看形体表面的顶点，并将顶点按逆时针方向编号，且由顶点坐标得到边向量，如图 6.2-2 所示。

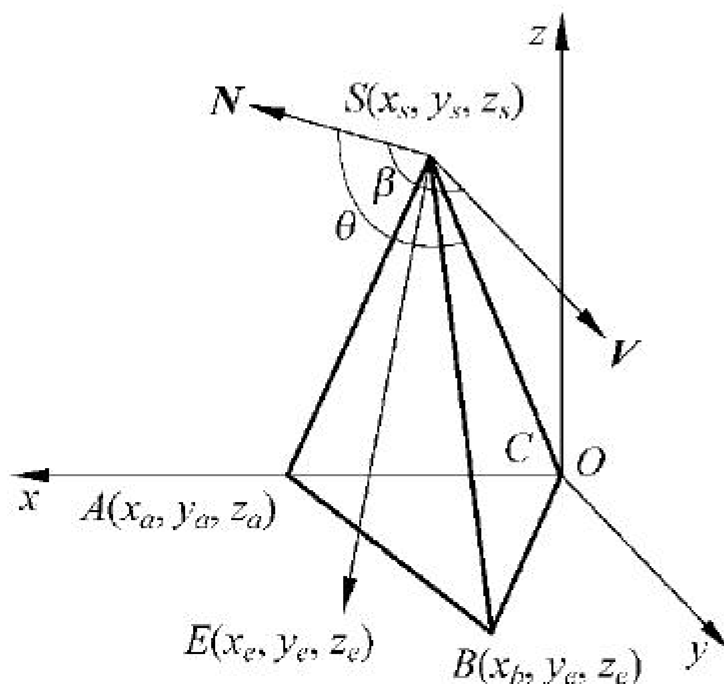


图 6.2-2 平面外法线向量

边向量$\overrightarrow{SA}$：从顶点 S 指向顶点 A。

边向量$\overrightarrow{AB}$：从顶点 A 指向顶点 B。

表面 SAB 的外法线向量 $\boldsymbol{N}=\overrightarrow{SA}\times\overrightarrow{AB}$，其方向指向表面外，两向量的向量积的坐标表示法线向量：

$$\boldsymbol{N}=\overrightarrow{SA}\times\overrightarrow{SB}=\begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ x_a-x_s & y_a-y_s & z_a-z_s \\ x_b-x_a & y_b-y_a & z_b-z_a \end{bmatrix}$$

$$=D\boldsymbol{i}+E\boldsymbol{j}+F\boldsymbol{k}=(D,E,F)$$

其中

$$D=\begin{vmatrix} y_a-y_s & z_a-z_s \\ y_b-y_a & z_b-z_a \end{vmatrix},\quad E=\begin{vmatrix} z_a-z_s & x_a-x_s \\ z_b-z_a & x_b-x_a \end{vmatrix},\quad F=\begin{vmatrix} x_a-x_s & y_a-y_s \\ x_b-x_a & y_b-y_a \end{vmatrix}$$

分别为 $\boldsymbol{N}$ 在 x 轴、y 轴、z 轴上的分量。

设 $\boldsymbol{V}$ 是从观察点到平面上任一点的连线所得的向量，为方便起见，通常取平面外法线 $\boldsymbol{N}$ 的起点为视线向量 $\boldsymbol{V}$ 的起点。设观察点为 $E(x_e,y_e,z_e)$，外法线向量的起点为 $S(x_s,y_s,z_s)$，则视线向量 $\boldsymbol{V}$ 为

$$\boldsymbol{V}=(x_e,y_e,z_e)-(x_s,y_s,z_s)=(x_e-x_s,y_e-y_s,z_e-z_s)$$

由向量 $\boldsymbol{N}$ 和 $\boldsymbol{V}$ 的数量积 $\boldsymbol{N}\cdot\boldsymbol{V}=|\boldsymbol{N}|\cdot|\boldsymbol{V}|\cos\theta$，得 $\cos\theta=\dfrac{\boldsymbol{N}\cdot\boldsymbol{V}}{|\boldsymbol{N}|\cdot|\boldsymbol{V}|}$，其正负号与 $\boldsymbol{N}\cdot\boldsymbol{V}$ 一致。

显然表面可见性取决于表面外法线向量 $\boldsymbol{N}$ 与视线向量 $\boldsymbol{V}$ 之间的夹角 θ：

$0^\circ\leqslant\theta\leqslant90^\circ$，则该表面可见；

$90^\circ\leqslant\theta\leqslant180^\circ$，则该表面不可见。

根据 $\cos\theta$ 与 $\boldsymbol{N}\cdot\boldsymbol{V}$ 的正负号关系，可得表面可见性的判断依据为：

若 $\boldsymbol{N}\cdot\boldsymbol{V}>0$，则该表面可见；

若 $\boldsymbol{N}\cdot\boldsymbol{V}\leqslant0$，则该表面不可见。

在图 6.2-2 中，若观察点 E 在 y 轴正向无穷远处时，视线向量 $\boldsymbol{V}$ 平行于 y 轴，这时表面的可见性由外法线向量 $\boldsymbol{N}$ 与 y 轴正向的夹角 β 确定。

$\boldsymbol{N}$ 与 y 轴正向的夹角 β 的余弦为

$$\cos\beta=\frac{E}{|\boldsymbol{N}|}$$

这样，表面可见性判断依据简化为仅考虑外法线向量 $\boldsymbol{N}(D,E,F)$ 在 y 轴上的分量 E 的正、负号即可。

例 6.1 以图 6.2-2 为例，假设该三棱锥各顶点坐标为 $S(1,1,2)$，$A(2,0,0)$，$B(1,2,0)$，$C(0,0,0)$，视线向量平行于 y 轴，判断△SAB、△SBC、△SCA 表面的可见性。

解 只需计算出各表面的外法线向量在 y 轴上的分量 E，根据其正负即可判断该表面的可见性。

△SAB 表面：

$$E=\begin{vmatrix} z_a-z_s & x_a-x_s \\ z_b-z_a & x_b-x_a \end{vmatrix}=\begin{vmatrix} 0-2 & 2-1 \\ 0-0 & 1-2 \end{vmatrix}=2>0,\quad \triangle SAB\text{ 表面可见}$$

$\triangle SBC$ 表面：

$$E=\begin{vmatrix} z_b-z_s & x_b-x_s \\ z_c-z_b & x_c-x_b \end{vmatrix}=\begin{vmatrix} 0-2 & 1-1 \\ 0-0 & 0-1 \end{vmatrix}=2>0,\quad \triangle SBC \text{ 表面可见}$$

$\triangle SCA$ 表面：

$$E=\begin{vmatrix} z_c-z_s & x_c-x_s \\ z_a-z_c & x_a-x_c \end{vmatrix}=\begin{vmatrix} 0-2 & 0-1 \\ 0-0 & 2-0 \end{vmatrix}=-4<0,\quad \triangle SCA \text{ 表面不可见}$$

需要注意：物体某一表面上各顶点的排列顺序，必须为观察者面对该表面各顶点按逆时针走向排列。

在编程实现消隐时，由于显示器屏幕所在平面为 xOy，与屏幕平面垂直且向里的方向为 z 方向，视线向量与 z 方向相反，故在判断表面可见性时，应以该表面外法线向量 $\boldsymbol{N}$ 在 z 轴的分量 F 的正负来判断：当 $F<0$ 时，该表面可见；$F>0$ 时，该表面不可见。

当多面体某个表面可见时，则绘制该表面的每个棱边的投影；不可见时，则不绘制该表面的投影。

对于第5章构造的平面线框拉伸体，当拉伸形成的是一个凸多面体时，可以利用上述方法实现该凸多面体的消隐。消隐时，首先判断上下表面的可见性，然后再判断每个侧面的可见性。由于在形成拉伸体时，上下两个多边形的内外环方向已经按照顺时针或者逆时针的方向进行了排序，所以在计算每个面的外法线向量时，直接顺序利用三个顶点进行计算即可。

在编程时，首先创建对话框来设置是否消隐的标识符 hideFlag，重写第5章的 DrawBody()函数，使其既可绘制线框图，也可以绘制消隐图。代码如下：

```
void DrawBody(CDC * pDC,CBody_Stretch &m_Body,COLORREF &m_DrawColor,double m_Matrix_V[][4],
CBody_Stretch Body[],int bodyNum = 0,int lineWidth = 0,int lineType = 0,bool hideFlag = false){
                                                        //拉伸实体
    if(hideFlag == false){                              //线框图
        DrawBodyOfLineFrame(pDC,m_Body,m_DrawColor,m_Matrix_V, lineWidth,lineType);
    }
    else{//消隐图
DrawBodyOfHideLine(pDC,m_Body,m_DrawColor,m_Matrix_V,Body,bodyNum,lineWidth,lineType);
    }
}
```

其中的拉伸线框函数 DrawBodyOfLineFrame()即为第5章的 DrawBody()函数。消隐函数 DrawBodyOfHideLine()代码如下：

```
/*********************************************************************************
    DrawBodyOfHideLine: 三维拉伸凸多面体在显示器屏幕上的消隐算法
pDC: 显示器指针; m_Body: 拾取的实体; m_Matrix_V: 透视变换矩阵; Body[]: 拉伸实体数组;
bodyNum: 实体数量; m_DrawColor: 颜色; lineWidth: 线宽; lineType: 线型
    *******************************************************************************/
    void DrawBodyOfHideLine(CDC * pDC,CBody_Stretch &m_Body,COLORREF &m_DrawColor,double m_
Matrix_V[][4],CBody_Stretch Body[],int bodyNum = 0,int lineWidth = 0,int lineType = 0){
        CPoint pt0_1,pt1_1,pt0_2,pt1_2;
        CPoint3D pt_3D,pt0_3D,pt1_3D,pt2_3D;
        //首先判断上下两个表面多边形的可见性(一个可见,则另外一个不可见)
```

```
int m = 0;     // = 0 为第一个表面可见,1 为第二个表面可见, - 1 则法矢 Z = 0,都不可见
pt0_3D = m_Body.EdgePlane[0].loop_out.GetAt(0).Pt1_3D;
pt1_3D = m_Body.EdgePlane[0].loop_out.GetAt(0).Pt2_3D;
pt2_3D = m_Body.EdgePlane[0].loop_out.GetAt(1).Pt2_3D;
double N_Z = VectorXVectorForZ(pt0_3D,pt1_3D,pt2_3D);
if(N_Z == 0)
    m = - 1;
else if(N_Z < 0){
    if(m_Body.length > 0)     //Z 正向拉伸,此时法线向量为内法线向量,故该面不可见,
                              //则拉伸的另外一个面可见
        m = 1;
    else
        m = 0;
}
else
    if(m_Body.length > 0)
        m = 0;
    else
        m = 1;
if(m!= - 1){                                  //画上下表面中可见表面
    for(int i = 0;i < m_Body.EdgePlane[m].loop_out.GetSize();i++){
        pt_3D = m_Body.EdgePlane[m].loop_out.GetAt(i).Pt1_3D;
        GetNewPoint(pt_3D,m_Matrix_V);
        pt0_1 = pt_3D.To2DPt();
        pt_3D = m_Body.EdgePlane[m].loop_out.GetAt(i).Pt2_3D;
        GetNewPoint(pt_3D,m_Matrix_V);
        pt0_2 = pt_3D.To2DPt();
        MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);      //画线
    }
}
//顺序判断每一个侧面
for(int k = 0;k < m_Body.EdgePlane[0].loop_out.GetSize();k++){
    pt0_3D = m_Body.EdgePlane[0].loop_out.GetAt(k).Pt1_3D;
    pt1_3D = m_Body.EdgePlane[0].loop_out.GetAt(k).Pt2_3D;
    pt2_3D = m_Body.EdgePlane[1].loop_out.GetAt(k).Pt1_3D;
    if(m_Body.length > 0)                     //正向拉伸计算外法矢
            N_Z = VectorXVectorForZ(pt0_3D,pt1_3D,pt2_3D);
    else
        N_Z = VectorXVectorForZ(pt2_3D,pt1_3D,pt0_3D);
    if(N_Z < 0){//可见,画四条棱边
        GetNewPoint(pt0_3D,m_Matrix_V);
        pt0_1 = pt0_3D.To2DPt();
        GetNewPoint(pt1_3D,m_Matrix_V);
        pt0_2 = pt1_3D.To2DPt();
        GetNewPoint(pt2_3D,m_Matrix_V);
        pt1_1 = pt2_3D.To2DPt();
        MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);      //画线
        MIDPOINT_Line(pDC,pt0_1,pt1_1,m_DrawColor,lineWidth);      //画线
        pt1_3D = m_Body.EdgePlane[1].loop_out.GetAt(k).Pt2_3D;
```

```
                GetNewPoint(pt1_3D,m_Matrix_V);
                pt0_1 = pt1_3D.To2DPt();
                MIDPOINT_Line(pDC,pt0_1,pt0_2,m_DrawColor,lineWidth);      //画线
                MIDPOINT_Line(pDC,pt0_1,pt1_1,m_DrawColor,lineWidth);      //画线
            }
        }
    }
```

其中，计算多边形的法线向量的 z 坐标即分量 F 的函数代码为：

```
double VectorXVectorForZ(CPoint3D &pt0,CPoint3D &pt1,CPoint3D &pt2){
    //计算向量的叉乘 k
    /* V = (xB - xA)·(y - yA) - (x - xA)·(yB - yA)
    (pt1.x - pt0.x) (pt1.y - pt0.y) 0
    (pt2.x - pt1.x) (pt2.y - pt1.y) 0 */
    return (pt1.x - pt0.x) * (pt2.y - pt1.y) - (pt2.x - pt1.x) * (pt1.y - pt0.y);
}
```

采用上述方法绘制的拉伸凸多面体消隐实例效果如图 6.2-3 所示。

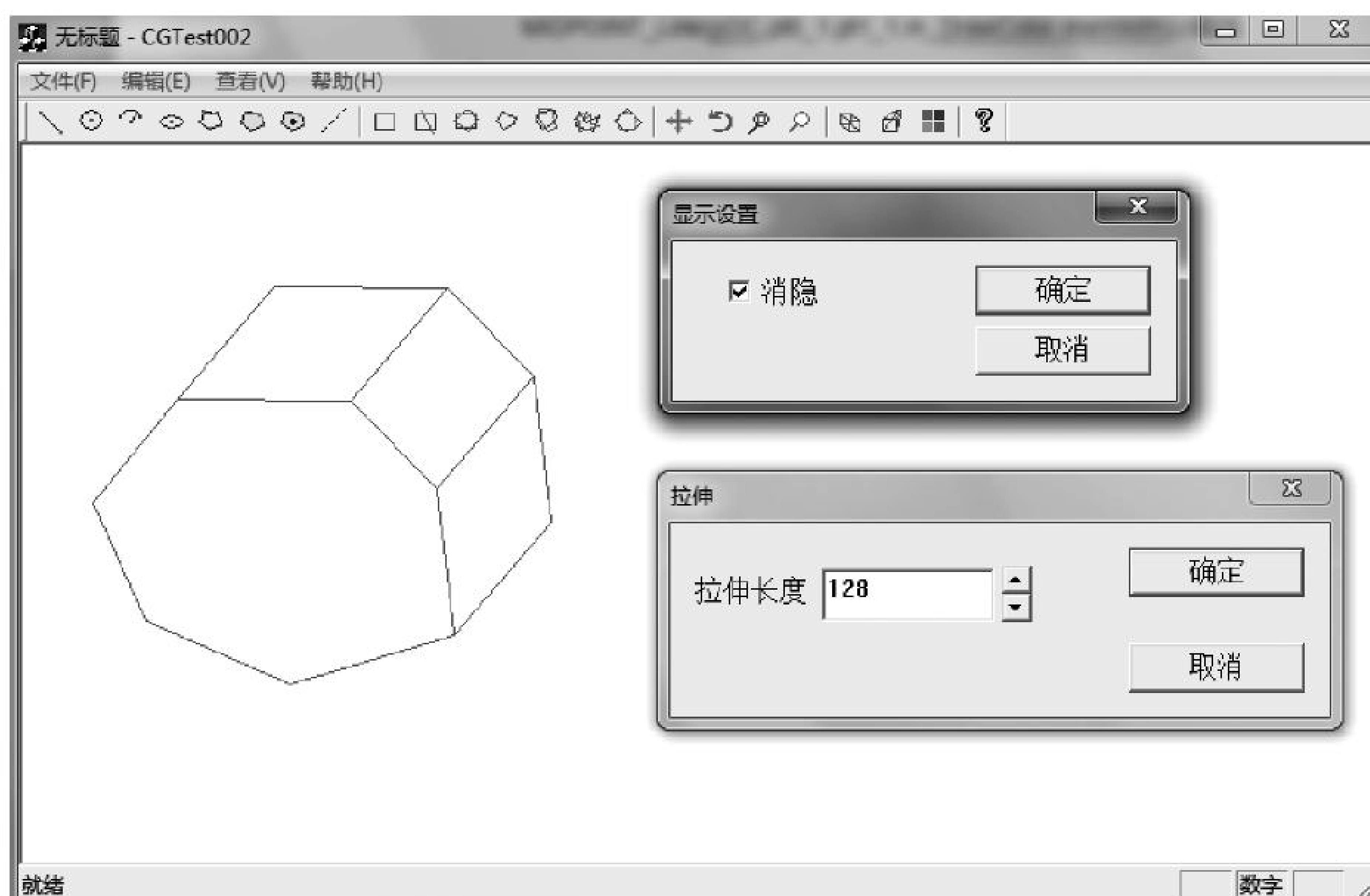


图 6.2-3　拉伸凸多面体消隐

凸多面体消隐处理的一般步骤可归纳如下。

步骤一，几何变换。

步骤二，计算各表面的外法线向量 $\mathbf{N}$。

步骤三，计算外法线向量 $\mathbf{N}$ 与视线向量 $\mathbf{V}$ 的夹角 θ 的余弦值 $\cos\theta$ 或判断其正负号。

步骤四，循环根据步骤三判断各表面的可见性。

步骤五，表面可见时(如赋值 1)，画出其平面多边形；不可见时(如赋值 0 或 -1)，不画出，处理下一个表面，直至最后一个表面。

6.3 一般多面体的消隐

6.3.1 消隐分析

一般多面体的表面可以是凸多边形、凹多边形，也可以是带内环的多边形。当多面体的某个表面在视线方向通过外法线向量判断为可见时，该表面可能最终只是部分可见，也有可能是完全不可见的，因此，对多面体上经外法线向量判断得出的可见面只能看作潜在可见面，其最终的可见情况还需做进一步的判定。

当空间有多个多面体时，多面体之间也需要判断遮挡关系，进行消隐处理。多面体之间表面的消隐判断和单个多面体表面之间的判断方法类似，因此，二者可以归结为同一类消隐问题。

一般多面体消隐算法的基本思路：首先，计算形体各表面外法线向量，将形体上的全部表面分为不可见面和潜在可见面；对于不可见面不再处理，而对于潜在可见面，再进一步判断其可见性。

各种消隐算法的区别主要在于对潜在可见面及其棱边的处理采用不同的思想方法实现。下面介绍几种经典的消隐算法，并对其中的扫描线消隐算法进行编程实现。

6.3.2 隐线算法

隐线算法的思想是：根据潜在可见面获得潜在可见棱线，依次从潜在可见棱线中取出一条棱线，其在显示屏幕的投影与非该棱线所在的其他潜在可见面的投影在深度（即 z 坐标）方向上进行判断比较，确定其是否被遮挡以及被遮挡的子线段的位置。

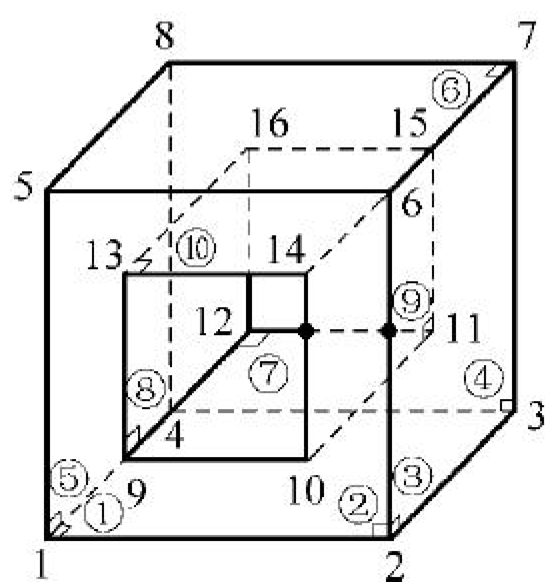


图 6.3-1　隐线算法示意图

如图 6.3-1 所示的多面体，在视线方向的潜在可见面分别为：1-2-6-5（内环 9-13-14-10）、2-3-7-6、5-6-7-8、9-12-16-13、9-10-11-12。

潜在可见面中顺序两个顶点以及首尾两个顶点的连线即为潜在可见棱线。依次取出一个潜在可见棱线，和其他潜在可见面进行遮挡关系比较。

例如，在潜在可见面 1-2-6-5（内环 9-13-14-10）中取棱线 1-2，和其他潜在可见面进行判断，由于均没有遮挡关系，则该棱线是可见的。

从潜在可见面 9-10-11-12 中，取棱线 11-12，与其他潜在可见面进行判断，在和潜在可见面 1-2-6-5（9-10-14-13）判断比较时，棱线 11-12 的中间线段⑨-⑦被遮挡，不可见，子线段 11-⑨和⑦-12 可见，在和潜在可见面 2-3-7-6 判断时，子线段 11-⑨被遮挡，不可见，最后只有⑦-12 可见。循环上述过程，即可确定每条潜在可见棱线的最终结果。

在进行线面遮挡关系判断时，算法可以按以下的步骤实现。

1. 包含性检测

利用边界盒检测法进行粗略的遮挡关系判断。边界盒是指包含表面投影的两对边分别平行于坐标轴的最小外接矩形，也称为最小投影矩形，如图 6.3-2 所示。如果空间两个图形的最小投影矩形没有重叠部分，则这两个图形相互之间不存在消隐问题。

2. 深度检测

首先，进行粗略的深度检测，找出对被测线段不构成遮挡的平面。在显示器屏幕上，如果被测的潜在可见棱线的两端点的 z 坐标都小于遮挡平面上每一顶点的 z 坐标，那么该遮挡平面不可能对这条棱线构成遮挡，可不必再作精确比较。

接着，进行精确的深度检测。

如图 6.3-3 所示，在该坐标系下，判断线段 $P_1(x_1, y_1, z_1)-P_2(x_2, y_2, z_2)$ 和遮挡平面的关系。线段 P_1P_2 在 y 轴投射方向上在平面上的对应线段是 $M_1(x_1, y_1', z_1)-M_2(x_2, y_2', z_2)$。

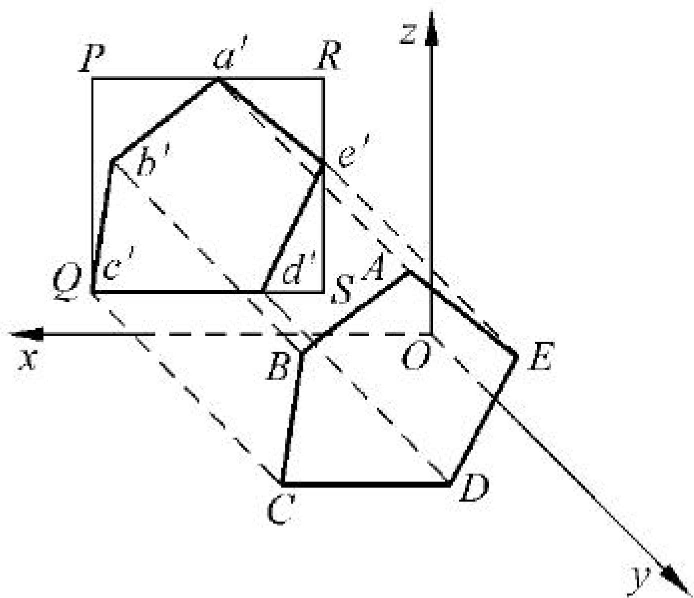


图 6.3-2　表面投影边界盒

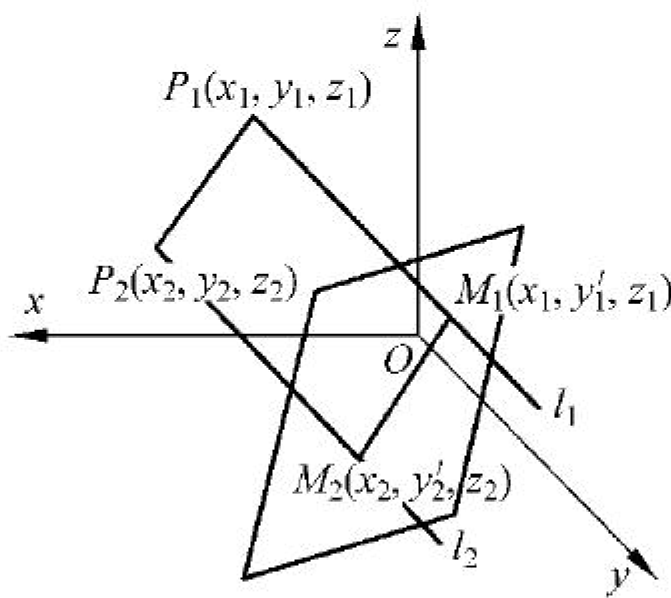


图 6.3-3　精确深度检测

比较两组对应点的 y 坐标值，有以下三种情况。

若 $y_1 > y_1'$ 且 $y_2 > y_2'$：棱线在遮挡面之前，棱线全部可见。

若 $y_1 < y_1'$ 且 $y_2 < y_2'$：棱线在遮挡面之后，棱线可能全部或部分被遮挡。

若 $y_1 < y_1'$ 且 $y_2 > y_2'$ 或 $y_1 > y_1'$ 且 $y_2 < y_2'$：棱线部分在遮挡面之前部分在遮挡面之后，棱线可能部分被遮挡。

通过上述分析也可获得被测线段两个端点的可见性。

因此，需要计算 y_1' 和 y_2'。对于 y_1'，令 $y_1' = y_1 + t_1$，设遮挡平面方程为

$$Ax + By + Cz + D = 0$$

则 t_1 值为 $t_1 = -(Ax_1 + By_1 + Cz_1 + D)/B$，由此即可得 y_1' 值。y_2' 值的计算同上。

平面方程的系数 A、B、C、D 可以这样计算：在潜在可见面上取逆时针走向三个相邻的顶点，坐标分别为 (x_1, y_1, z_1)、(x_2, y_2, z_2)、(x_3, y_3, z_3)，则

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = Ax + By + Cz + D = 0$$

$$A = y_1(z_2 - z_3) - y_2(z_1 - z_3) + y_3(z_1 - z_2)$$

$$B = -x_1(z_2 - z_3) + x_2(z_1 - z_3) - x_3(z_1 - z_2)$$

$$C = x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)$$

$$D = -x_1(y_2z_3 - y_3z_2) + x_2(y_1z_3 - y_3z_1) - x_3(y_1z_2 - y_2z_1)$$

3. 隐藏性判别和隐藏线消除

首先，判别隐藏性。如投影平面是 xOz 平面，则对被测棱线的投影与遮挡平面边框的各线段的投影进行求交运算，如图 6.3-4 所示。

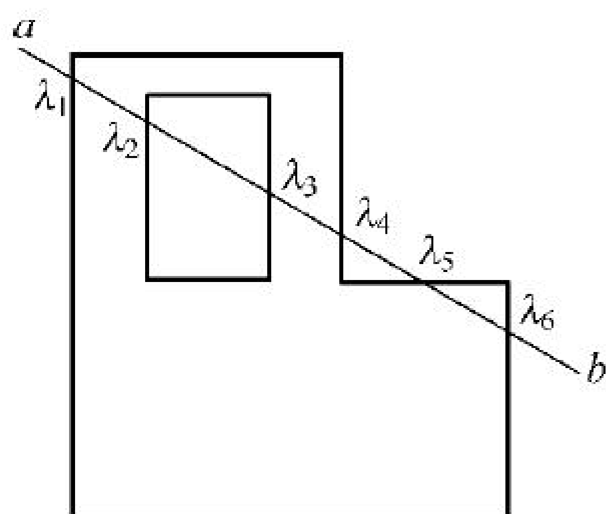


图 6.3-4 投影求交点

被测棱线在 xOz 平面上投影线段的参数方程为

$$\begin{cases} x = x_1 + (x_2 - x_1)\lambda \\ z = z_1 + (z_2 - z_1)\lambda \end{cases}$$

遮挡平面边框的各线段在 xOz 平面上的参数方程为

$$\begin{cases} x = x_{i1} + (x_{i2} - x_{i1})\mu \\ z = z_{i1} + (z_{i2} - z_{i1})\mu \end{cases}$$

联立两方程可依次求出各交点，并计算出其相应的 λ 和 μ 的值。当 $0 \leqslant \lambda \leqslant 1$ 和 $0 \leqslant \mu \leqslant 1$ 时，交点位于两线段上，为有效交点，否则为无效交点。该交点是被测棱线和遮挡平面边框投影的重影点，比较它们的对应深度值即 y 坐标的大小，确定该点在被测棱线投影的可见性，并在被测棱线投影线上按 λ 的大小对交点进行排序。

接着，进一步判断被交点分割的被测棱线的各子线段的可见性。若子线段的两个端点都是不可见的，则该子线段不可见，不再判断。只要子线段的其中一个端点是可见的，则该子线段即可见。对于可见的子线段，继续进行上述过程，直到与所有的潜在可见表面都进行了遮挡性判断，获得最终的可见子线段。

隐线算法主要适用于线框体的消隐，在消隐过程中基本不考虑物体的表面特性，因为只是进行线线求交和线面求交，每条线占用内存较小，所以运算速度较快，属于物体空间消隐算法。

6.3.3 画家算法

所谓画家算法是根据画家作画形成一幅图画的过程构造的算法。画家画油画时先画远景，再画中间景物，最后才画近景。画家通过这种顺序构造画面，自然地解决了可见性问题。因此，画家算法的基本思路如下。

首先，把屏幕置成背景色。

接着，将场景中的物体按其距观察点的远近进行排序，结果放在一张线性表中。(线性表构造方法：距观察点远的优先级低，放在表头；距观察点近的优先级高，放在表尾。该表称为深度优先级表)

然后，按照从远到近(从表头到表尾)的顺序逐个绘制物体。优先级低的先画，优先级高的后画，优先级高的表面颜色取代优先级低的表面颜色，相当于消除了隐藏面。

画家算法的关键是对场景中的物体按深度(远近)进行优先级排序，并在图像空间内进行消隐，它又称为深度排序算法，属于物体图像空间算法。

一种对表面多边形的排序算法如下。

步骤一，将场景中所有多边形存入一个线性表，记为 L。

步骤二，如果 L 中仅有一个多边形，算法结束；否则根据每个多边形的 z_{min} 对它们预排

序。假定多边形 P 落在表首，即 $z_{\min}(P)$ 为最小。记 Q 为 L 中除去 P 的任意一个多边形。

步骤三，判别 Q、P 之间的关系，有如下两种情况。

(1) 对所有的 Q，有 $z_{\max}(P)<z_{\min}(Q)$，则多边形 P 距观察点最远，它不可能遮挡别的多边形。令 $L=L-P$，即去掉 P，返回步骤二。

(2) 若存在某一个多边形 Q，使 $z_{\max}(P)>z_{\min}(Q)$，需进一步判别。

① 若 P、Q 的投影 P'、Q'的包围盒不相交，则 P、Q 在表中的次序不重要，令 $L=L-P$，返回步骤二；否则进行下一步。

② 若 P 的所有顶点位于 Q 所在平面的不可见的一侧，则 P、Q 关系正确，令 $L=L-P$，返回步骤二；否则进行下一步。

③ 若 Q 的所有顶点位于 P 所在平面的可见的一侧，则 P、Q 关系正确，令 $L=L-P$，返回步骤二；否则进行下一步。

④ 对 P、Q 的投影 P'、Q'求交。P'、Q'不相交，则 P、Q 在表中的次序不重要，令 $L=L-P$，返回步骤二。否则在它们所相交的区域中任取一点，计算 P、Q 在该点的深度值，如果 P 的深度小，则 P、Q 关系正确，令 $L=L-P$，返回步骤二；否则交换 P、Q，返回步骤三。

上述算法不能处理多边形循环遮挡和多边形相互穿透的情况，解决办法是将多边形分割成多个多边形再处理。

画家算法主要适用于面消隐的情况。由于画家算法对多边形集合中的每个点都要进行渲染，而没有考虑这些多边形在最终场景中可能被其他部分遮挡，因此，此种算法效率较低，对于细致的场景来说，会过度地消耗计算机资源。

6.3.4 深度缓冲器算法

深度缓冲器算法又称 Z-Buffer 算法，可以看作是画家算法的一个发展，它根据每个像素的信息解决深度冲突的问题，并且抛弃了对于深度渲染顺序的依赖。

深度缓冲器算法的基本思路是：对于显示屏上的每一个像素，记录下位于该像素内最靠近观察者的那个景物面的深度坐标，同时相应记录下用来显示该景物面的颜色(或亮度信息)，那么所有记录下的这些像素所对应的颜色就可以形成最后要输出的图形。

深度缓冲器算法的主要问题是需要定义两个巨大的数组：一个是深度数组，即 Z 缓冲器 depth[x][y]；另一个是颜色或亮度数组，即帧缓冲器 intensity[x][y]。其中的 x、y 分别表示显示屏幕上沿 x 轴方向和沿 y 轴方向上分布的像素数目。帧缓冲器与 Z 缓冲器的单元一一对应，如图 6.3-5 所示。

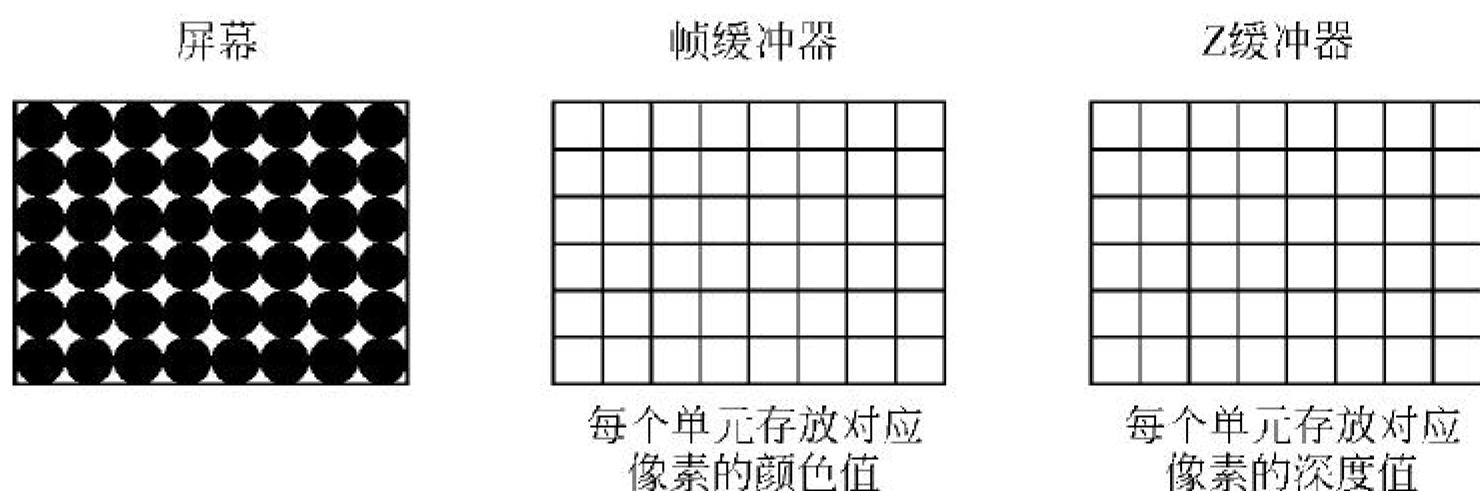


图 6.3-5 Z-Buffer 算法中屏幕、帧与 Z 缓冲器单元一一对应

Z-Buffer 算法对形体的每一个潜在可见面全部进行采样，并将采样点变换到屏幕坐标系的图像空间，记录其深度坐标值 z。根据采样点在屏幕上投影的位置，将其深度与已存储的 Z 缓冲器里的深度值进行比较。如果新采样点的深度值大于原存储值，则用新深度值替换原有的深度值，同时在帧缓冲器中替换相应的单元的颜色或者亮度值。

Z-Buffer 算法是所有图像空间算法中最简单的一种隐藏面消除算法。它在像素级上以近物取代远物，与形体在屏幕上的出现顺序无关。其优点是：简单稳定，利于硬件实现，不需要整个场景的几何数据。缺点是：对每个多边形占据的每个像素都要计算深度值，计算量大，并占用大量的内存空间，编程实现时也发现 Z-Buffer 算法对计算机资源耗费较多，在实时图形变换时，图形刷新与其他算法相比有明显延迟现象。

6.3.5 基于扫描线的消隐算法

由于 Z-Buffer 算法需要较大的内存资源，为克服这个缺点，可以将整个绘图区域分割成若干个小区域，然后一个区域一个区域地显示，这样 Z 缓冲器的单元数只要等于一个区域内像素的个数就可以了。如果将小区域取成屏幕上的扫描线，就得到扫描线 Z 缓冲器算法。

扫描线 Z 缓冲器算法对于扫描线上的每个像素点还要计算深度值，甚至不止一次的计算，运算量仍然很大。改进为，在一条扫描线上，每个区间只计算一次深度，即区间扫描线算法，又称扫描线算法。

当消隐的目的只是消除形体被遮挡的部分，而不考虑表面的光照颜色及亮度时，那么，扫描线算法只对潜在可见棱边进行处理即可，这时计算所需的内存将进一步减少。与多边形的扫描转换类似，当用一条扫描线扫描形体的投影时，计算其与形体某一个潜在可见面的所有潜在可见棱边投影的交点，该交点的数据结构中需要记录其在对应棱边上的深度即 z 坐标信息，然后在扫描线方向上将交点按增量进行排序，并两两组成交点区间对，则组成区间对内的点属于潜在可见面的投影。当交点对集合中插入新的交点对时，需判断新老交点对在空间对应的棱边部分的遮挡关系，在对交点区间连线进行遮挡关系比较时，如两条线在深度方向存在包含、部分遮挡等关系，需将被遮挡部分分解为可见的多个交点对。

如图 6.3-6 所示，一条扫描线与形体的潜在可见面投影相交情况如下：

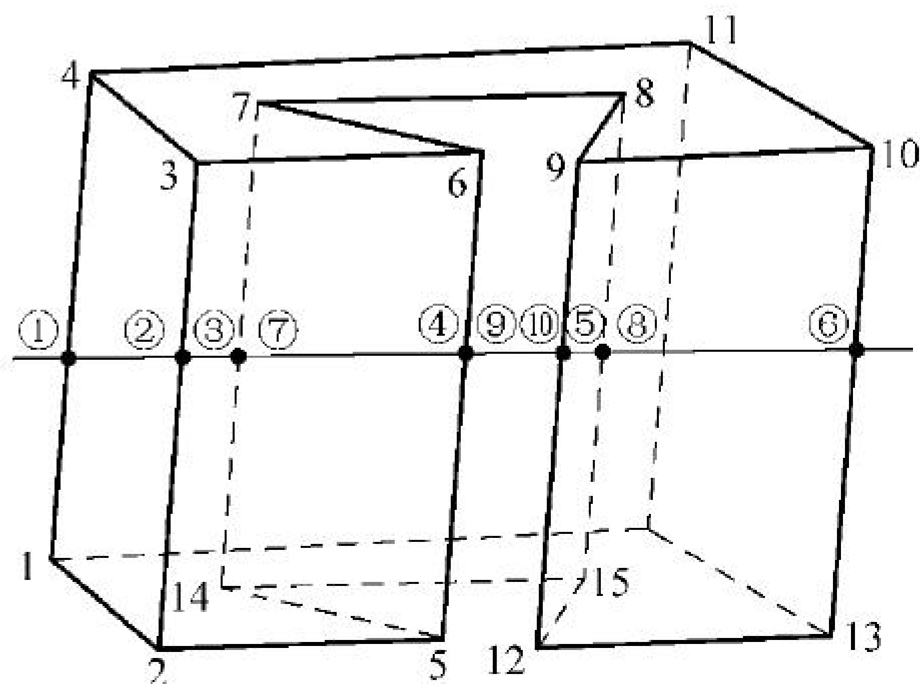


图 6.3-6 扫描线消隐法

与面 1-2-3-4 的交点是①和②；

与面 2-5-6-3 的交点是③和④，从图中可以看出②和③是重合的；

与面 12-13-10-9 的交点是⑤和⑥；

与面 14-15-8-7 的交点是⑦和⑧。

由于还需要判断深度信息，所以在交点的数据结构中要记录交点在视线方向对应于所在面的 z 坐标值大小。

在扫描线 x 方向上对交点进行排序，由于前面三个面的交点的 x 坐标值递增，则交点对的集合及交点区间对顺序为①-②、③-④、⑤-⑥。当扫描线与面 14-15-8-7 的交点对⑦-⑧插入交点对集合中时，顺序判断其与集合每个交点对的 x 坐标值大小，首先与①-②比较，由于⑦比②的 x 坐标值大，则判断下一组③-④。由于⑦比④小、⑧比④大，所以两组交点对的连线在视线方向存在遮挡关系。

首先，计算⑦-⑧连线上与④重影的点在面 14-15-8-7 上的 z 坐标值：

$$t = (x_4 - x_7)/(x_8 - x_7)$$

则

$$z = z_7 + (z_8 - z_7)t$$

比较 z 与 z_4，显然④可见，⑦-⑧中⑦-④部分被遮挡，将⑦-⑧中与④重影的点⑨记录下来，则⑨-⑧段是⑦-⑧未被③-④遮挡的部分。

下面继续比较⑨-⑧与下一组交点对⑤-⑥的遮挡关系。和上面的方法类似，判断⑤-⑥连线上与⑧重影的点在面 12-13-10-9 上的 z 坐标值，显然⑧被遮挡，在⑨-⑧上取与⑤重影的点⑩，则⑨-⑩是⑨-⑧中未被遮挡的部分，并插在⑤-⑥交点对之前。

最后，即得该扫描线与潜在可见平面相交的可见交点对集合及顺序为①-②、③-④、⑨-⑩、⑤-⑥。

从上面分析可以看出，在对交点对排序时，需要详细判断和处理待插入交点对与当前被比较的交点对之间的位置关系以及遮挡关系。设待插入交点对为③-④，当前用于比较的交点对为①-②，则两组交点对的位置和遮挡关系可以分为如图 6.3-7 所示的 6 种情况。

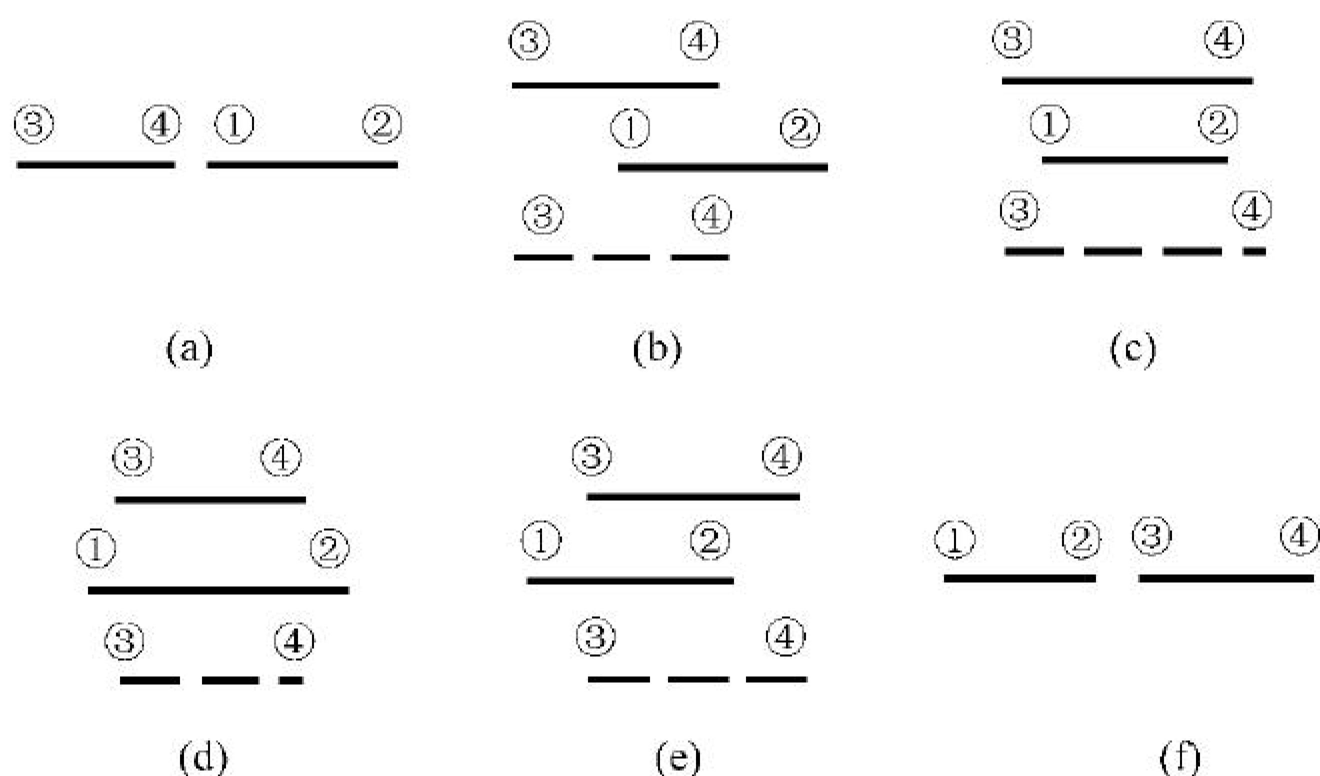


图 6.3-7 交点对之间的相对位置及遮挡关系

在图 6.3-7 中，除了(a)和(f)外，其他几种情况下的两组交点对之间均存在遮挡关系，其中，虚线表示在对应位置下存在遮挡关系的情况。当存在遮挡时，需要对被遮挡的部分分

成可见段和不可见段来处理。

上述方法，当对交点区间内的像素点用所对应表面的颜色显示时，即可实现形体表面的渲染，如果只对交点用给定的颜色显示，那么获得的就是形体边的消隐。

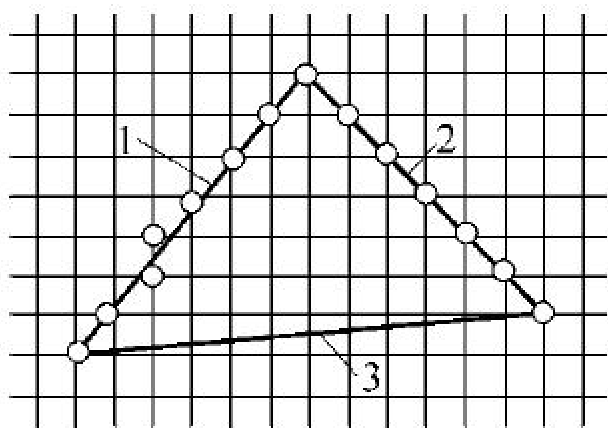


图 6.3-8 扫描线特殊情况

由于扫描线法主要是利用扫描线与边投影的交点来实现消隐，但是，并没有进一步分析边投影的连贯性特点，那么，扫描线与同一个边投影所有交点的集合有可能并不连续。如图 6.3-8 所示，利用扫描线法扫描三个边，对边 1、边 2 的扫描均可获得同一个边的连续性逼近交点，但是对于边 3 扫描线扫描时，只能获得边投影的两个端部的交点，不能获得边投影的整个连续的部分。当边的投影直线斜率较小，即边比较水平时，扫描线法获得的边的这种像素点不连续性会一直存在。

为了使扫描线法获得的边是连贯的，需要在不连续的像素点之间进行补偿使之连续，这需要进一步判断相邻扫描线获得的同一边的像素交点之间是否连续，如不连续，则在两个交点之间再进行连线。这样，在交点的数据结构中，除了需要登记深度坐标信息外，还需登记所在边的信息。为了实现边的连贯性，对于扫描线上被其他交点对遮挡隐藏的交点，在插入交点对排序时，不要删除，仍然将其插入交点集合中，但是设置其属性为不可见，在判断相邻扫描线的边连续性时再使用。

则投影交点的数据结构类可参考如下：

```
class CPixelPt{
public:
    int x,y;
    double z;             //深度坐标 z 值
    COLORREF EdgeColor;  //如是边界点时,边界的颜色
    COLORREF DrawColor;  //如是内部点时,内部点的颜色
    int SeeFlag;          ///是否可见,0: 不可见,1: 可见
    int InterFlag;        //是否是扫描线交点,1: 是,0: 不是
    int Type;             ///点类型,0: 普通点,1: 边界点,2: 边界顶点,3: 边界是水平线时的顶点
    CEdge Edge[2];        //投影像素点所在边
public:
    CPixelPt(){
        SeeFlag = 1;      //默认点是可见的
        Type = 0;         //默认是普通点
        InterFlag = 1;    //默认是交点
    }
    CPixelPt& operator = (const CPixelPt &PixelPt){
        if(this ==  &PixelPt)
        {return * this;}
        else{
            x = PixelPt.x;
            y = PixelPt.y;
            z = PixelPt.z;
            EdgeColor = PixelPt.EdgeColor;
            DrawColor = PixelPt.DrawColor;
            Type = PixelPt.Type;
            InterFlag = PixelPt.InterFlag;
```

```
                SeeFlag = PixelPt.SeeFlag;
                Edge[0] = PixelPt.Edge[0];
                Edge[1] = PixelPt.Edge[1];
                return *this;
            }
        }
};
```

消隐主要是对实体表面的潜在可见性进行分析，所以，也需要建立实体表面的数据结构，可参考如下：

```
class CBodySurf:CObject{
public:
    CEdgePlane Body_Surf;                //表面内外环，已排序
    BOOL BodySurf_See_Flag;              //-1：不可见，0：潜在可见，1：确定可见
    int y_min_Project,y_max_Project;     //该表面在xOy面的投影的扫描线的最大最小值
    COLORREF color;                      //表面颜色
    int PickFlag;                        //是否拾取，0：没有，1：拾取
    CBodySurf(){
        BodySurf_See_Flag = FALSE;
        y_min_Project = 0;
        y_max_Project = 0;
        color = RGB(255,255,255);
        PickFlag = 0;
    }
    ~CBodySurf(){
    }
    CBodySurf& operator = (CBodySurf &BodySurf){
        if(this == &BodySurf)
            return *this;
        else{
            this->Body_Surf = BodySurf.Body_Surf;
            this->BodySurf_See_Flag = BodySurf.BodySurf_See_Flag;
            this->y_min_Project = BodySurf.y_min_Project;
            this->y_max_Project = BodySurf.y_max_Project;
            this->color = BodySurf.color;
            this->PickFlag = BodySurf.PickFlag;
            return *this;
        }
    }
    CBodySurf(const CBodySurf& BodySurf){
        this->BodySurf_See_Flag = BodySurf.BodySurf_See_Flag;
        this->y_min_Project = BodySurf.y_min_Project;
        this->y_max_Project = BodySurf.y_max_Project;
        this->color = BodySurf.color;
        this->PickFlag = BodySurf.PickFlag;
    }
};
```

基于扫描线的消隐算法的步骤可以归纳如下。

第一步：计算出所有实体表面的总数量 BodySurf_num，创建实体表面数组 Body_Surf[]，分析每个表面的潜在可见性，占有的扫描线的最大最小位置，以及该表面是否是被拾取的实体的表面。如有表面颜色，也可设置表面颜色。

第二步：从所有表面的最小扫描线 $y_{scan}=y_{min}$（如果 $y_{scan}\leqslant y_{max}$，$i=0$）开始，如果 $i<$ BodySurf_num，从实体表面数组中顺序取出一个潜在可见表面 Body_Surf[i]，判断其与 y_{scan}是否相交。如 y_{scan}与 Body_Surf[i]相交，则计算交点，获得交点集合 m_OneSurf_Pt_Array。

第三步：将交点集合 m_OneSurf_Pt_Array 的新交点两两组合插入该扫描线原有的交点集合 m_All_Pt_Array 中，按 x 坐标大小排序，并分析处理交点对之间的遮挡关系；如果 $i<$BodySurf_num，则 $i=i+1$，返回第二步。

第四步：依次从交点集合 m_All_Pt_Array 中取出一个交点 PixelPt0，判断其是否可见。如不可见，但是其为计算得到的交点，则判断其与上一条扫描线产生的交点集合中同一边的交点之间是否连续。如不连续则连线，重复循环第四步。

第五步：如交点 PixelPt0 可见，再判断下一个交点是否和它在同一个水平边上。如是，直接连线，返回第四步。

第六步：判断点 PixelPt0 与上一条扫描线产生的交点集合中同一边的交点之间是否连续，如不连续则连线。然后设置 PixelPt0 的颜色，返回第四步。

上述步骤的参考代码如下（其中的三维实体是第 5 章三维造型中的线框拉伸实体）：

```
/*********************************************************************
    DrawTrueBody: 三维拉伸凸多面体在显示器屏幕上的消隐算法
pDC: 显示器指针; m_Picker: 拾取的实体; m_EdgeColor: 拾取时的颜色; m_Matrix_V: 透视变换矩阵; Body[]: 拉伸实体数组; bodyNum: 实体数量; m_DrawColor: 未拾取的颜色
    *********************************************************************/
    void DrawTrueBody(CDC * pDC,CPicker &m_Picker,COLORREF m_EdgeColor,double m_Matrix_V[]
[4],CBody_Stretch Body[],COLORREF &m_DrawColor,int bodyNum){
        int i,j,x,y,m,n,k,s;      //第一步创建实体表面,首先计算一共有多少实体表面
        int BodySurf_num = 0;                       //表面数量
        CBodySurf bodySurf;
        for(m = 0;m < bodyNum;m++){
            BodySurf_num += 2;
            BodySurf_num += Body[m].EdgePlane[0].loop_out.GetSize();     //外环
            for(int i = 0;i < Body[m].EdgePlane[0].in_num;i++){
                BodySurf_num += Body[m].EdgePlane[0].loop_in[i].GetSize();
            }
        }
        CBodySurf * Body_Surf = new CBodySurf[BodySurf_num];     //设置实体表面数组
        bool flag = FALSE;
        //1.创建实体表面数组,并获得最小最大扫描线
        s = 0;
        int pickFlag = 0;
        int yscan,ymin = 0,ymax = 0;
        for(m = 0;m < bodyNum;m++){
            CPoint3D pt_3D[5];                      //侧面多边形
```

```
CEdge Edge;
CArray<CEdge,CEdge>m_Edge_Array;
if(m_Picker.picktype==pick_body&&Body[m]==m_Picker.m_Body_Stretch)
    pickFlag=1;
else
    pickFlag=0;
if(Body[m].length>0){                    //下表面反向
    bodySurf.Body_Surf=Body[m].EdgePlane[0];
    bodySurf.Body_Surf.AntiDirection();//反向
    bodySurf.color=Body[m].color;        //设置表面颜色
    BuildBodySurf(bodySurf);             //计算该表面的可见性及占有的扫描线等
    bodySurf.PickFlag=pickFlag;          //设置该表面的拾取状态
    Body_Surf[s++]=bodySurf;             //加入表面数组
    if(m==0){//计算最小最大扫描线
        ymin=bodySurf.y_min_Project;
        ymax=bodySurf.y_max_Project;
    }
    else{
        if(bodySurf.BodySurf_See_Flag==1){
            if(bodySurf.y_min_Project<ymin) ymin=bodySurf.y_min_Project;
            if(bodySurf.y_max_Project>ymax) ymax=bodySurf.y_max_Project;
        }
    }
    bodySurf.Body_Surf=Body[m].EdgePlane[1];     //上表面
    bodySurf.color=Body[m].color;
    BuildBodySurf(bodySurf);
    bodySurf.PickFlag=pickFlag;
    Body_Surf[s++]=bodySurf;
    if(bodySurf.BodySurf_See_Flag==1){
        if(bodySurf.y_min_Project<ymin) ymin=bodySurf.y_min_Project;
        if(bodySurf.y_max_Project>ymax) ymax=bodySurf.y_max_Project;
    }
    for(int i=0;i<Body[m].EdgePlane[0].loop_out.GetSize();i++){//外环侧面
        m_Edge_Array.RemoveAll();        //作为表面外环,按逆时针方向排序
        pt_3D[0]=Body[m].EdgePlane[0].loop_out.GetAt(i).Pt1_3D;  //取一个边
        pt_3D[1]=Body[m].EdgePlane[0].loop_out.GetAt(i).Pt2_3D;
        pt_3D[2]=Body[m].EdgePlane[1].loop_out.GetAt(i).Pt2_3D;
        pt_3D[3]=Body[m].EdgePlane[1].loop_out.GetAt(i).Pt1_3D;
        pt_3D[4]=Body[m].EdgePlane[0].loop_out.GetAt(i).Pt1_3D;
        for(int j=0;j<4;j++){
            Edge.Pt1_3D=pt_3D[j];
            Edge.Pt2_3D=pt_3D[j+1];
            m_Edge_Array.Add(Edge);
        }
        bodySurf.Body_Surf.loop_out.RemoveAll();
        bodySurf.Body_Surf.loop_out.Append(m_Edge_Array);     //加入表面数组
        bodySurf.Body_Surf.in_num=0;
        bodySurf.color=Body[m].color;
        BuildBodySurf(bodySurf);
        bodySurf.PickFlag=pickFlag;
        Body_Surf[s++]=bodySurf;
```

```
            if(bodySurf.BodySurf_See_Flag == 1){
                if(bodySurf.y_min_Project < ymin) ymin = bodySurf.y_min_Project;
                if(bodySurf.y_max_Project > ymax) ymax = bodySurf.y_max_Project;
            }
        }
        for(k = 0;k < Body[m].EdgePlane[0].in_num;k++){      //内环侧表面
            for(n = 0;n < Body[m].EdgePlane[0].loop_in[k].GetSize();n++){
                m_Edge_Array.RemoveAll();
                //作为表面外环,按逆时针方向排序
                pt_3D[0] = Body[m].EdgePlane[0].loop_in[k].GetAt(n).Pt1_3D;
                pt_3D[1] = Body[m].EdgePlane[0].loop_in[k].GetAt(n).Pt2_3D;
                pt_3D[2] = Body[m].EdgePlane[1].loop_in[k].GetAt(n).Pt2_3D;
                pt_3D[3] = Body[m].EdgePlane[1].loop_in[k].GetAt(n).Pt1_3D;
                pt_3D[4] = Body[m].EdgePlane[0].loop_in[k].GetAt(n).Pt1_3D;
                for(int j = 0;j < 4;j++){
                    Edge.Pt1_3D = pt_3D[j];
                    Edge.Pt2_3D = pt_3D[j + 1];
                    m_Edge_Array.Add(Edge);
                }
                bodySurf.Body_Surf.loop_out.RemoveAll();
                bodySurf.Body_Surf.loop_out.Append(m_Edge_Array);
                bodySurf.Body_Surf.in_num = 0;
                bodySurf.color = Body[m].color;
                BuildBodySurf(bodySurf);
                bodySurf.PickFlag = pickFlag;
                Body_Surf[s++] = bodySurf;
            }
        }
    }
    else{     //下表面反向
        bodySurf.Body_Surf = Body[m].EdgePlane[0];
        bodySurf.color = Body[m].color;
        BuildBodySurf(bodySurf);
        bodySurf.PickFlag = pickFlag;
        Body_Surf[s++] = bodySurf;
        if(m == 0){
            ymin = bodySurf.y_min_Project;
            ymax = bodySurf.y_max_Project;
        }
        else{
            if(bodySurf.BodySurf_See_Flag == 1){
                if(bodySurf.y_min_Project < ymin) ymin = bodySurf.y_min_Project;
                if(bodySurf.y_max_Project > ymax) ymax = bodySurf.y_max_Project;
            }
        }
        //上表面
        bodySurf.Body_Surf = Body[m].EdgePlane[1];
        bodySurf.Body_Surf.AntiDirection();     //反向
        bodySurf.color = Body[m].color;
        BuildBodySurf(bodySurf);
        bodySurf.PickFlag = pickFlag;
```

```
        Body_Surf[s++] = bodySurf;
        if(bodySurf.BodySurf_See_Flag == 1){
            if(bodySurf.y_min_Project < ymin) ymin = bodySurf.y_min_Project;
            if(bodySurf.y_max_Project > ymax) ymax = bodySurf.y_max_Project;
        }
        for(i = 0;i < Body[m].EdgePlane[0].loop_out.GetSize();i++){  //外环侧表面
            m_Edge_Array.RemoveAll();       //作为表面外环,按逆时针方向排序
pt_3D[0] = Body[m].EdgePlane[0].loop_out.GetAt(i).Pt2_3D; //逆时针取外环的一个边
            pt_3D[1] = Body[m].EdgePlane[0].loop_out.GetAt(i).Pt1_3D;
            pt_3D[2] = Body[m].EdgePlane[1].loop_out.GetAt(i).Pt1_3D;
            pt_3D[3] = Body[m].EdgePlane[1].loop_out.GetAt(i).Pt2_3D;
            pt_3D[4] = Body[m].EdgePlane[0].loop_out.GetAt(i).Pt2_3D;
            for(int j = 0;j < 4;j++){
                Edge.Pt1_3D = pt_3D[j];
                Edge.Pt2_3D = pt_3D[j + 1];
                m_Edge_Array.Add(Edge);
            }
            bodySurf.Body_Surf.loop_out.RemoveAll();
            bodySurf.Body_Surf.loop_out.Append(m_Edge_Array);  //加入表面数组
            bodySurf.Body_Surf.in_num = 0;
            //BuildBodySurf(bodySurf);
            bodySurf.color = Body[m].color;
            BuildBodySurf(bodySurf);
            bodySurf.PickFlag = pickFlag;
            Body_Surf[s++] = bodySurf;
            if(bodySurf.BodySurf_See_Flag == 1){
                if(bodySurf.y_min_Project < ymin) ymin = bodySurf.y_min_Project;
                if(bodySurf.y_max_Project > ymax) ymax = bodySurf.y_max_Project;
            }
        }
        for(int k = 0;k < Body[m].EdgePlane[0].in_num;k++){  //内环侧表面
            for(int n = 0;n < Body[m].EdgePlane[0].loop_in[k].GetSize();n++){
                m_Edge_Array.RemoveAll();
                //作为表面外环,按逆时针方向排序
                pt_3D[0] = Body[m].EdgePlane[0].loop_in[k].GetAt(n).Pt2_3D;
                pt_3D[1] = Body[m].EdgePlane[0].loop_in[k].GetAt(n).Pt1_3D;
                pt_3D[2] = Body[m].EdgePlane[1].loop_in[k].GetAt(n).Pt1_3D;
                pt_3D[3] = Body[m].EdgePlane[1].loop_in[k].GetAt(n).Pt2_3D;
                pt_3D[4] = Body[m].EdgePlane[0].loop_in[k].GetAt(n).Pt2_3D;
                for(int j = 0;j < 4;j++){
                    Edge.Pt1_3D = pt_3D[j];
                    Edge.Pt2_3D = pt_3D[j + 1];
                    m_Edge_Array.Add(Edge);
                }
                bodySurf.Body_Surf.loop_out.RemoveAll();
                bodySurf.Body_Surf.loop_out.Append(m_Edge_Array);    //加入数组
                bodySurf.Body_Surf.in_num = 0;
                bodySurf.color = Body[m].color;
                BuildBodySurf(bodySurf);
                bodySurf.PickFlag = pickFlag;
                Body_Surf[s++] = bodySurf;
```

```
                }
            }
        }
    }
    //2.从最小扫描线开始,取出一个表面判断交点
    CArray<CPixelPt,CPixelPt> m_All_Pt_Array,m_OneSurf_Pt_Array, m_All_Pt_Array_Old;
                                            //m_All_Pt_Array_Old记录上一次的交点集
    if(ymin<0)ymin=0;
    int height= GetSystemMetrics ( SM_CYSCREEN );//获得当前屏幕的高度位置
    if(ymax>height)ymax=height;
    CPixelPt PixelPt0,PixelPt,PixelPt_0,PixelPt_1;
    CPoint startPoint,endPoint;
    for(yscan=ymin;yscan<=ymax;yscan++){
        m_All_Pt_Array.RemoveAll();
        for(s=0;s<BodySurf_num;s++){
if(Body_Surf[s].BodySurf_See_Flag==TRUE&&yscan>=Body_Surf[s].y_min_Project&&yscan<=
Body_Surf[s].y_max_Project)                          //判断是否可见和是否相交
        //扫描该面,获得交点对
        m_OneSurf_Pt_Array.RemoveAll();
        if(Body_Surf[s].PickFlag==1)    //是拾取实体的表面,扫描该表面获得交线集合
            ScanProjectSurf(yscan,Body_Surf[s],m_EdgeColor,m_OneSurf_Pt_Array);
        else                                         //扫描该表面获得交线集合
            ScanProjectSurf(yscan,Body_Surf[s],m_DrawColor,m_OneSurf_Pt_Array);
        InsertPtArray(m_All_Pt_Array,m_OneSurf_Pt_Array);     //插入m_All_Pt_Array
        }
        //消隐显示该扫描线
        for(j=0;j<m_All_Pt_Array.GetSize();j++){
            PixelPt0=m_All_Pt_Array.GetAt(j); //取一个交点
            if(PixelPt0.SeeFlag==0)//&&PixelPt.SeeFlag==0){      //不可见
                if(PixelPt0.InterFlag==1)//判断它与上一条扫描线的交点是否需要连线
                LinkPixelPt(pDC,PixelPt0,j,m_All_Pt_Array,m_All_Pt_Array_Old);
                    continue;
            }
            for(k=j+1;k<m_All_Pt_Array.GetSize();k++){
                PixelPt=m_All_Pt_Array.GetAt(k);
                if(PixelPt.SeeFlag==0)
                    continue;
                else{
                    //判断两个点是否在同一个水平边上,如在,直接连接起来
if(abs(PixelPt0.x-PixelPt.x)>2&&(PixelPt.Edge[0]==PixelPt0.Edge[0]||PixelPt.Edge[0]==
PixelPt0.Edge[1]||PixelPt.Edge[1]==PixelPt0.Edge[0]||PixelPt.Edge[1]==PixelPt0.Edge
[1])){
                    startPoint.x=PixelPt0.x;
                    startPoint.y=PixelPt0.y;
                    endPoint.x=PixelPt.x;
                    endPoint.y=PixelPt.y;
                    MIDPOINT_Line(pDC,startPoint,endPoint,PixelPt0.EdgeColor);
                    break;
                }
                else
                    break;
```

```
                }
            }
            //判断和上一条扫描线的同一个边的交点之间是否有连线
            LinkPixelPt(pDC,PixelPt0,j,m_All_Pt_Array,m_All_Pt_Array_Old);
            //绘制交点,为了醒目,绘制相邻三个像素点
            pDC->SetPixel(PixelPt0.x,PixelPt0.y,PixelPt0.EdgeColor);
            pDC->SetPixel(PixelPt0.x-1,PixelPt0.y,PixelPt0.EdgeColor);
            pDC->SetPixel(PixelPt0.x+1,PixelPt0.y,PixelPt0.EdgeColor);
        }
        m_All_Pt_Array_Old.RemoveAll();
        m_All_Pt_Array_Old.Append(m_All_Pt_Array);  //将当前交点记录下来
    }
}
```

上述代码中,计算某实体表面的可见性及占有的扫描线的函数代码如下:

```
void BuildBodySurf(CBodySurf& BodySurf){
    CPoint3D pt0_3D,pt1_3D,pt2_3D,pt_limit;
    int yi;
    //首先判断表面是否潜在可见,利用表面的外环点判断,用极值点判断,例如在y方向的极值
点,找到极值点对应的i
    int i_edge=0;
    pt_limit=BodySurf.Body_Surf.loop_out.GetAt(0).Pt1_3D;
    for(int i=1;i<BodySurf.Body_Surf.loop_out.GetSize();i++){
        if(BodySurf.Body_Surf.loop_out.GetAt(i).Pt1_3D.y>pt_limit.y){
            pt_limit=BodySurf.Body_Surf.loop_out.GetAt(i).Pt1_3D;
            i_edge=i;
        }
    }
    if(i_edge==0)
pt0_3D=BodySurf.Body_Surf.loop_out.GetAt(BodySurf.Body_Surf.loop_out.GetSize()-1).Pt1_3D;
    else
        pt0_3D=BodySurf.Body_Surf.loop_out.GetAt(i_edge-1).Pt1_3D;
    pt1_3D=pt_limit;
    pt2_3D=BodySurf.Body_Surf.loop_out.GetAt(i_edge).Pt2_3D;
    if(VectorXVectorForZ(pt0_3D,pt1_3D,pt2_3D)<0){ //矢量叉乘Z<0
        BodySurf.BodySurf_See_Flag=TRUE;            //设置潜在可见
        //计算投影扫描线的最大最小值
BodySurf.y_max_Project=BodySurf.y_min_Project=BodySurf.Body_Surf.loop_out.GetAt(0).Pt1_
3D.To2DPt().y;
    for(int i=1;i<BodySurf.Body_Surf.loop_out.GetSize();i++){
        yi=BodySurf.Body_Surf.loop_out.GetAt(i).Pt1_3D.To2DPt().y;
        if(yi>BodySurf.y_max_Project)BodySurf.y_max_Project=yi;
        else if(yi<BodySurf.y_min_Project) BodySurf.y_min_Project=yi;
        }
    }
    else{
        BodySurf.BodySurf_See_Flag=FALSE;            //设置不可见
BodySurf.y_max_Project=BodySurf.y_min_Project=BodySurf.Body_Surf.loop_out.GetAt(0).Pt1_
3D.To2DPt().y;
        for(int i=1;i<BodySurf.Body_Surf.loop_out.GetSize();i++){
```

```
                yi = BodySurf.Body_Surf.loop_out.GetAt(i).Pt1_3D.To2DPt().y;
                if(yi > BodySurf.y_max_Project)BodySurf.y_max_Project = yi;
                else if(yi < BodySurf.y_min_Project) BodySurf.y_min_Project = yi;
            }
        }
    }
```

计算扫描线与实体表面投影相交的函数如下：

```
/* yi: 当前扫描线; BodySurf: 当前表面; m_EdgeDrawColor: 交点颜色; m_Pt_Array: 交点集合 */
    void ScanProjectSurf(int& yi,CBodySurf& BodySurf,COLORREF& m_EdgeDrawColor,CArray<CPixelPt,
CPixelPt>& m_Pt_Array){
        //循环从内外环中取边
        int m_x;                                    //交点
        double z;
        CLine line;
        CArray<CEdge,CEdge> m_Edge_Array;
        int i,j,k;
        CPoint Pt_0;
        CArray<CPoint,CPoint> m_VPt_Array;          //已处理的顶点集合
        double t;
        int Type;                                   //交点类型
        m_Pt_Array.RemoveAll();
        for(int m = 0;m < BodySurf.Body_Surf.in_num + 1;m++){
            m_Edge_Array.RemoveAll();
            if(m == 0)                              //首先扫描外环投影
                m_Edge_Array.Append(BodySurf.Body_Surf.loop_out);
            else                                    //扫描内环投影
                m_Edge_Array.Append(BodySurf.Body_Surf.loop_in[m - 1]);
            for(i = 0;i < m_Edge_Array.GetSize();i++){
                //判断每条边是否和扫描线相交,如相交,求交点,插入交点集并排序
    if(yi == (int)(m_Edge_Array.GetAt(i).Pt1_3D.y + 0.5)&&yi == (int)(m_Edge_Array.GetAt
(i).Pt2_3D.y + 0.5)){//是水平线,则将两个端点加入点集
                z = m_Edge_Array.GetAt(i).Pt1_3D.z;
                    Type = 3;
    OrderToInsertPt_x(m_Pt_Array,m_Edge_Array.GetAt(i).Pt1_3D.x + 0.5,yi,z,m_
EdgeDrawColor,BodySurf.color,Type,m_Edge_Array.GetAt(i),m_Edge_Array.GetAt(i == 0?m_Edge_
Array.GetSize() - 1:i - 1));
    OrderToInsertPt_x(m_Pt_Array,m_Edge_Array.GetAt(i).Pt2_3D.x + 0.5,yi,z,m_
EdgeDrawColor,BodySurf.color,Type,m_Edge_Array.GetAt(i),m_Edge_Array.GetAt(i == m_Edge_
Array.GetSize() - 1?0:i + 1));
            }
            else
if((yi - (int)(m_Edge_Array.GetAt(i).Pt1_3D.y + 0.5)) * (yi - (int)(m_Edge_Array.GetAt(i).
Pt2_3D.y + 0.5))< 0)//||(yi >= m_Edge_Array.GetAt(i).Pt2_3D.y&&yi <= m_Edge_Array.GetAt(i).
Pt1_3D.y)){
            //求交点
    m_x = GetInterPtXForScanY(yi,(int)(m_Edge_Array.GetAt(i).Pt1_3D.x + 0.5),(int)(m_Edge_
Array.GetAt(i).Pt1_3D.y + 0.5),(int)(m_Edge_Array.GetAt(i).Pt2_3D.x + 0.5),(int)(m_Edge_
Array.GetAt(i).Pt2_3D.y + 0.5));
            //交点排序,首先计算交点处的 z 坐标
```

```
        t = ((double)(m_x - m_Edge_Array.GetAt(i).Pt1_3D.x))/(m_Edge_Array.GetAt(i).Pt2_
3D.x - m_Edge_Array.GetAt(i).Pt1_3D.x);
        z = m_Edge_Array.GetAt(i).Pt1_3D.z + (m_Edge_Array.GetAt(i).Pt2_3D.z - m_Edge_
Array.GetAt(i).Pt1_3D.z) * t;                    //找到表面对应的边在投影交点处的 z 值
        Type = 1;                                //交点取一个
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i));
        }
        else if((yi - (int)(m_Edge_Array.GetAt(i).Pt1_3D.y + 0.5)) == 0){
            z = m_Edge_Array.GetAt(i).Pt1_3D.z;
            m_x = m_Edge_Array.GetAt(i).Pt1_3D.x + 0.5;
                if(i == 0){
    if((yi - (int)(m_Edge_Array.GetAt(i).Pt2_3D.y + 0.5)) * (yi - (int)(m_Edge_Array.GetAt(m
_Edge_Array.GetSize() - 1).Pt1_3D.y + 0.5))> 0){
    if(CheckVPt(m_VPt_Array,m_x,yi) == 0){      //该交点没有处理,再加一个交点
        Type = 2;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(m_Edge_Array.GetSize() - 1));
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(m_Edge_Array.GetSize() - 1),m_Edge_Array.GetAt(i));
        }
    }
    else if((yi - (int)(m_Edge_Array.GetAt(i).Pt2_3D.y + 0.5)) * (yi - (int)(m_Edge_Array.
GetAt(m_Edge_Array.GetSize() - 1).Pt1_3D.y + 0.5))< 0){  //只加一个交点
        Type = 1;
        if(CheckVPt(m_VPt_Array,m_x,yi) == 0)         //没有处理
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(m_Edge_Array.GetSize() - 1));
        }
        else{                                     //只加一个交点
            Type = 1;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(m_Edge_Array.GetSize() - 1));
            }
        }
        else{
    if((yi - (int)(m_Edge_Array.GetAt(i).Pt2_3D.y + 0.5)) * (yi - (int)(m_Edge_Array.GetAt(i
 - 1).Pt1_3D.y + 0.5))> 0){                    //判断该交点是否已处理过
        if(CheckVPt(m_VPt_Array,m_x,yi) == 0){     //没有处理,再加一个交点
            Type = 2;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i - 1));
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i - 1),m_Edge_Array.GetAt(i));
            }
        }
        else
if((yi - (int)(m_Edge_Array.GetAt(i).Pt2_3D.y + 0.5)) * (yi - (int)(m_Edge_Array.GetAt(i -
1).Pt1_3D.y + 0.5))< 0){//只加一个交点
        Type = 1;
        if(CheckVPt(m_VPt_Array,m_x,yi) == 0)         //没有处理
```

```
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i-1));
        }
        else{                                       //只加一个交点
            Type=1;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i-1));
            }
        }
    }
    else if((yi-(int)(m_Edge_Array.GetAt(i).Pt2_3D.y+0.5))==0){
        z=m_Edge_Array.GetAt(i).Pt2_3D.z;
        m_x=m_Edge_Array.GetAt(i).Pt2_3D.x+0.5;
        if(i==m_Edge_Array.GetSize()-1){
    if((yi-(int)(m_Edge_Array.GetAt(0).Pt2_3D.y+0.5))*(yi-(int)(m_Edge_Array.GetAt
(i).Pt1_3D.y+0.5))>0){                              //判断该交点是否已处理过
        if(CheckVPt(m_VPt_Array,m_x,yi)==0){        //没有处理,再加一个交点
            Type=2;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(0));
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(0),m_Edge_Array.GetAt(i));
                                    }
                                }
                                else
if((yi-(int)(m_Edge_Array.GetAt(0).Pt2_3D.y+0.5))*(yi-(int)(m_Edge_Array.GetAt(i).
Pt1_3D.y+0.5))<0){                                  //只加一个交点
            Type=1;
            if(CheckVPt(m_VPt_Array,m_x,yi)==0)     //没有处理
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(0));
            }
            else{                                   //只加一个交点
                Type=1;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(0));
                }
            }
            else{
    if((yi-(int)(m_Edge_Array.GetAt(i+1).Pt2_3D.y+0.5))*(yi-(int)(m_Edge_Array.
GetAt(i).Pt1_3D.y+0.5))>0){                         //判断该交点是否已处理过
            if(CheckVPt(m_VPt_Array,m_x,yi)==0){    //没有处理,再加一个交点
                Type=2;
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i+1));
    OrderToInsertPt_x(m_Pt_Array,m_x,yi,z,m_EdgeDrawColor,BodySurf.color,Type,m_Edge_
Array.GetAt(i+1),m_Edge_Array.GetAt(i));
            }
        }
    else
if((yi-(int)(m_Edge_Array.GetAt(i+1).Pt2_3D.y+0.5))*(yi-(int)(m_Edge_Array.GetAt
```

```
(i).Pt1_3D.y + 0.5))< 0){                                //只加一个交点
        Type = 1;
        if(CheckVPt(m_VPt_Array,m_x,yi) == 0)            //没有处理
    OrderToInsertPt_x(m_Pt_Array, m_x, yi, z, m_EdgeDrawColor, BodySurf.color, Type, m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i + 1));
        }
        else{                                            //只加一个交点
            Type = 1;
    OrderToInsertPt_x(m_Pt_Array, m_x, yi, z, m_EdgeDrawColor, BodySurf.color, Type, m_Edge_
Array.GetAt(i),m_Edge_Array.GetAt(i + 1));
            }
        }    }  }  }
    }
```

其中，判断该交点是否在点集的函数代码为：

```
int CheckVPt(CArray < CPoint,CPoint > & m_VPt_Array, int &m_x, int &yi){
    CPoint pt;
    pt.x = m_x;
    pt.y = yi;
    for(int i = 0;i < m_VPt_Array.GetSize();i++){
        if(pt == m_VPt_Array.GetAt(i))
            return 1;
    }
    m_VPt_Array.Add(pt);
    return 0;
}
```

在交点集合中加入新点并排序的函数代码如下：

```
void OrderToInsertPt_x(CArray < CPixelPt,CPixelPt > & m_PixelPt_Array, int m_x, int& yi,double&
z,COLORREF& m_EdgeColor,COLORREF& m_DrawColor,int& Type,CEdge& edge1,CEdge& edge2){
    CPixelPt PixelPt,PixelPt0,PixelPt1,PixelPt2;
    PixelPt.x = m_x;
    PixelPt.y = yi;
    PixelPt.EdgeColor = m_EdgeColor;
    PixelPt.DrawColor = m_DrawColor;
    PixelPt.z = z;
    PixelPt.Type = Type;
    PixelPt.Edge[0] = edge1;
    PixelPt.Edge[1] = edge2;
    for(int i = 0;i < m_PixelPt_Array.GetSize();i++){
        PixelPt0 = m_PixelPt_Array.GetAt(i);
        if(m_x < PixelPt0.x){
            m_PixelPt_Array.InsertAt(i,PixelPt);
            return;
        }
        else if(PixelPt0.x == m_x){
            if(PixelPt0.Type == 3){
                //检查邻近的一个点是不是水平线的另外一个交点
                if(i!= 0){
                    PixelPt1 = m_PixelPt_Array.GetAt(i - 1);
```

```
                if(PixelPt1.Type == 3&&PixelPt1.Edge[0] == PixelPt0.Edge[0]){
                    m_PixelPt_Array.InsertAt(i + 1,PixelPt);  //插在后面
                    return;
                }
            }
            if(i < m_PixelPt_Array.GetSize() - 1){
                PixelPt1 = m_PixelPt_Array.GetAt(i + 1);
                if(PixelPt1.Type == 3&&PixelPt1.Edge[0] == PixelPt0.Edge[0]){
                    m_PixelPt_Array.InsertAt(i,PixelPt);      //插在前面
                    return;
                }
            }
            m_PixelPt_Array.InsertAt(i,PixelPt);              //插在前面
                return;
        }
        else if(PixelPt.Type == 3){                           //插入水平边界的点
            //检查另外一点是否已经插入
            if(i!= 0){
                PixelPt1 = m_PixelPt_Array.GetAt(i - 1);
                if(PixelPt1.Type == 3&&PixelPt1.Edge[0] == PixelPt.Edge[0]){
                    m_PixelPt_Array.InsertAt(i,PixelPt);      //插在当前点前面
                    return;
                }
                else if(i - 2 >= 0){     //再判断前一点是否为水平边界的点
                    PixelPt2 = m_PixelPt_Array.GetAt(i - 2);
if(PixelPt2.Type == 3&&PixelPt2.Edge[0] == PixelPt.Edge[0]){
                        //PixelPt1 和 PixelPt2 须互换位置
                        m_PixelPt_Array.RemoveAt(i - 1);
                        m_PixelPt_Array.InsertAt(i - 2,PixelPt1);
                        m_PixelPt_Array.InsertAt(i,PixelPt); //插在当前点前面
                        return;
                    }
                }
            }
            if(i < m_PixelPt_Array.GetSize() - 1){
                PixelPt1 = m_PixelPt_Array.GetAt(i + 1);
                if(PixelPt1.Type == 3&&PixelPt1.Edge[0] == PixelPt.Edge[0]){
                    m_PixelPt_Array.InsertAt(i + 1,PixelPt);  //插在当前点前面
                    return;
                }
                else if(i + 2 < m_PixelPt_Array.GetSize()){
                    PixelPt2 = m_PixelPt_Array.GetAt(i + 2);
if(PixelPt2.Type == 3&&PixelPt2.Edge[0] == PixelPt.Edge[0]){
                        //PixelPt1 和 PixelPt2 须互换位置
                        m_PixelPt_Array.RemoveAt(i + 2);
                        m_PixelPt_Array.InsertAt(i + 1,PixelPt2);
                    m_PixelPt_Array.InsertAt(i + 1,PixelPt);  //插在当前点后面
                        return;
                    }
                }
            }
```

```
                m_PixelPt_Array.InsertAt(i,PixelPt);              //插在前面
                return;
            }
        }
    }
    m_PixelPt_Array.Add(PixelPt);                                 //交点值大,则加入尾部
}
```

扫描线新获得的交点集合插入原有交点集合,即将新的交点对插入原有交点对的函数代码如下:

```
void InsertPtArray(CArray<CPixelPt,CPixelPt>& m_All_Pt_Array,CArray<CPixelPt,CPixelPt>&
m_OneSurf_Pt_Array){
    CPixelPt Pt_1,Pt_2,Pt_3,Pt_4,Pt_Tmp;
    double t = 0,z,t1,z1;
    int flag = 0;                          ///是否处理标识,0: 未处理,1: 处理
    int SeeFlag,InterFlag,Type;
    int i,j,k,j0,j1;                       //j0 是现在的 Pt_1 下标位置,j1 为 Pt_2 的下标位置
    for(i = 0;i <= m_OneSurf_Pt_Array.GetSize() - 2;i++,i++){
        Pt_3 = m_OneSurf_Pt_Array.GetAt(i);
        Pt_4 = m_OneSurf_Pt_Array.GetAt(i + 1);
        flag = 0;                          //是否处理标识
        for(j = 0;j <= m_All_Pt_Array.GetSize() - 2;j++,j++){
            Pt_1 = m_All_Pt_Array.GetAt(j);
            j0 = j;
            if(Pt_1.SeeFlag == 0){
                for(k = j + 1;k < m_All_Pt_Array.GetSize() - 1;k++){
                    Pt_1 = m_All_Pt_Array.GetAt(k);
                    if(Pt_1.SeeFlag == 0)
                        continue;
                    else{
                        j0 = j = k;        //j 循环时,从 k + 1 开始
                        break;
                    }
                }
            }
            Pt_2 = m_All_Pt_Array.GetAt(j + 1);
            j1 = j + 1;
            if(Pt_2.SeeFlag == 0){         //不可见,循环下一个
                for(k = j + 2;k < m_All_Pt_Array.GetSize();k++){
                    Pt_2 = m_All_Pt_Array.GetAt(k);
                    if(Pt_2.SeeFlag == 0)
                        continue;
                    else{
                        j = k - 1;         //j 循环时,从 k + 1 开始
                        j1 = k;
                        break;
                    }
                }
            }
            if(Pt_2.x <= Pt_3.x)
```

```
                        continue;
                    else if(Pt_4.x<=Pt_1.x){      //插入该区间对之前
                        m_All_Pt_Array.InsertAt(j0,Pt_3);
                        m_All_Pt_Array.InsertAt(j0+1,Pt_4);
                        flag=1;
                        break;
                    }
                    else if(Pt_3.x<Pt_1.x&&Pt_4.x>Pt_2.x){
                        //比较直线 Pt_3、Pt_4 和 Pt2 重影的空间点的 z
                        t=((double)(Pt_2.x-Pt_3.x))/(Pt_4.x-Pt_3.x);
                        z=(Pt_4.z-Pt_3.z)*t+Pt_3.z;
                        if(z<Pt_2.z){             //Pt_1-Pt_2 不可见,Pt_3 插入 Pt_1 之前
                            m_All_Pt_Array.InsertAt(j0,Pt_3);
                            //Pt_1,Pt_2 都不可见,不删除,设置为不可见
                            m_All_Pt_Array.RemoveAt(j0+1);
                            Pt_1.SeeFlag=0;
                            m_All_Pt_Array.InsertAt(j0+1,Pt_1);
                            m_All_Pt_Array.RemoveAt(j1+1);
                            Pt_2.SeeFlag=0;
                            m_All_Pt_Array.InsertAt(j1+1,Pt_2);
                            //在 Pt_2.x 处构造新 Pt_3,并循环
                            SeeFlag=0;            //设置为不可见,和前面的真正的 Pt_3 联系
                            InterFlag=0;          //非原始交点
                            Type=0;               //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_2.EdgeColor,Pt_2.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);
                            //循环下一区间
                            j++;
                            continue;
                        }
                        else if(z>Pt_2.z){
                            //Pt_1-Pt_2 可见,Pt_3-Pt_4 被遮挡,部分不可见
                            //Pt_3 及 Pt_4 被分成两段,构造新的 Pt_3,继续判断新的 Pt_3-Pt_4
                            m_All_Pt_Array.InsertAt(j0,Pt_3);     //Pt_3 插入 Pt_1 之前
                            t=((double)(Pt_1.x-Pt_3.x))/(Pt_4.x-Pt_3.x);
                            z=(Pt_4.z-Pt_3.z)*t+Pt_3.z;
                            SeeFlag=1;            //可见,构成点对
                            InterFlag=0;          //非交点
                            Type=0;               //普通点
CreateNewPt(Pt_Tmp,Pt_1.x,Pt_1.y,z,Pt_4.EdgeColor,Pt_4.DrawColor,SeeFlag,InterFlag,Type,
Pt_1.Edge);
                            m_All_Pt_Array.InsertAt(j0+1,Pt_Tmp);     //插在 Pt_1 之前
                            //计算新 z 值
                            t=((double)(Pt_2.x-Pt_3.x))/(Pt_4.x-Pt_3.x);
                            z=(Pt_4.z-Pt_3.z)*t+Pt_3.z;
                            SeeFlag=1;            //可见,构成点对
                            InterFlag=0;          //非交点
                            Type=0;               //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_3.EdgeColor,Pt_3.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);
                            j++;
```

```
                j++;
                continue;
            }
            else if(z == Pt_2.z){
                //判断另外一个点 Pt_1
                t = ((double)(Pt_1.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                if(z < Pt_1.z){
                    //Pt_3 - Pt_4 可见,Pt_1 - Pt_2 不可见,Pt_3 插入 Pt_1 之前
                    m_All_Pt_Array.InsertAt(j0,Pt_3);
                    //Pt_1、Pt_2 都不可见,不删除,设置为不可见
                    m_All_Pt_Array.RemoveAt(j0 + 1);
                    Pt_1.SeeFlag = 0;
                    m_All_Pt_Array.InsertAt(j0 + 1,Pt_1);

                    m_All_Pt_Array.RemoveAt(j1 + 1);
                    Pt_2.SeeFlag = 0;
                    m_All_Pt_Array.InsertAt(j1 + 1,Pt_2);
                    //在 Pt_2.x 处构造新 Pt_3,并循环
                    SeeFlag = 0;        //不可见,构成点对
                    InterFlag = 0;      //非交点
                    Type = 0;           //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_3.EdgeColor,Pt_3.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);
                    //循环下一区间
                    j++;
                    continue;
                }
                else if(z > Pt_1.z){
                //Pt_3 及 Pt_4 被分成两段,构造新的 Pt_3,继续判断新的 Pt_3 - Pt_4
                    m_All_Pt_Array.InsertAt(j0,Pt_3);     //Pt_3 插入 Pt_1 之前
                    t = ((double)(Pt_1.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                    z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                    SeeFlag = 1;        //可见,构成点对
                    InterFlag = 0;      //非交点
                    Type = 0;           //普通点
CreateNewPt(Pt_Tmp,Pt_1.x,Pt_1.y,z,Pt_4.EdgeColor,Pt_4.DrawColor,SeeFlag,InterFlag,Type,
Pt_1.Edge);
                    m_All_Pt_Array.InsertAt(j0 + 1,Pt_Tmp);     //插在 Pt_1 之前
                    //计算新 z 值
                    t = ((double)(Pt_2.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                    z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                    SeeFlag = 1;        //可见,构成点对
                    InterFlag = 0;      //非交点
                    Type = 0;           //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_3.EdgeColor,Pt_3.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);

                    j++;
                    j++;
                    continue;
                }
```

```
                }
            }
            else if(Pt_3.x < Pt_1.x&&Pt_4.x <= Pt_2.x){
                //比较直线 Pt_1、Pt_2 和 Pt4 重影的空间点的 z
                t = ((double)(Pt_4.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                if(z < Pt_4.z){
//Pt_1 - Pt_2 可见,Pt_3 插入 Pt_1 之前,Pt_4 插入 Pt_1 之后,不可见,Pt_4 缩短插在 Pt_1 前
                    m_All_Pt_Array.InsertAt(j0,Pt_3);     //Pt_3 插入 Pt_1 之前
                    t = ((double)(Pt_1.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                    z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                    SeeFlag = 1;            //可见,构成点对
                    InterFlag = 0;          //非交点
                    Type = 0;               //普通点
CreateNewPt(Pt_Tmp,Pt_1.x,Pt_1.y,z,Pt_4.EdgeColor,Pt_4.DrawColor,SeeFlag,InterFlag,Type,
Pt_1.Edge);
                    m_All_Pt_Array.InsertAt(j0 + 1,Pt_Tmp);     //插在 Pt_1 之前
                    Pt_4.SeeFlag = 0;
                    m_All_Pt_Array.InsertAt(j0 + 3,Pt_4);     //Pt_4 不可见,插入 Pt_1 之后
                    flag = 1;
                    break;
                }
                else if(z > Pt_4.z){
                    //Pt_3 - Pt_4 可见,Pt_1 不可见(设置),并缩短到 Pt_4.x 的位置
                    m_All_Pt_Array.RemoveAt(j0);
                    Pt_1.SeeFlag = 0;
                    m_All_Pt_Array.InsertAt(j0,Pt_1);
                    m_All_Pt_Array.InsertAt(j0,Pt_3);        //Pt_3 插入 Pt_1 之前
                    m_All_Pt_Array.InsertAt(j0 + 2,Pt_4);     //Pt_4 插入 Pt_1 之后
                    //Pt_1 不可见,缩短到 Pt_4.x 的位置
                    //修改 Pt_1 的值,插入 Pt_4 之后
                    SeeFlag = 1;            //可见,构成点对
                    InterFlag = 0;          //非交点
                    Type = 0;               //普通点
CreateNewPt(Pt_Tmp,Pt_4.x,Pt_4.y,z,Pt_1.EdgeColor,Pt_1.DrawColor,SeeFlag,InterFlag,Type,
Pt_4.Edge);
                    m_All_Pt_Array.InsertAt(j0 + 3,Pt_Tmp);
                    flag = 1;
                    break;                  //插入完毕,循环下一个
                }
                else if(z == Pt_4.z){
                    //判断另外一个点 Pt_1
                    t = ((double)(Pt_1.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                    z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                    if(z < Pt_1.z){
                    //Pt_3 - Pt_4 可见,Pt_1 不可见(设置),并缩短到 Pt_4.x 的位置
                        m_All_Pt_Array.RemoveAt(j0);
                        Pt_1.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j0,Pt_1);
                        m_All_Pt_Array.InsertAt(j0,Pt_3);        //Pt_3 插入 Pt_1 之前
                        m_All_Pt_Array.InsertAt(j0 + 2,Pt_4);     //Pt_4 插入 Pt_1 之后
```

```
                        //Pt_1 不可见,缩短到 Pt_4.x 的位置
                        //修改 Pt_1 的值,插入 Pt_4 之后
                        SeeFlag = 1;        //可见,构成点对
                        InterFlag = 0;      //非交点
                        Type = 0;           //普通点
CreateNewPt(Pt_Tmp,Pt_4.x,Pt_4.y,Pt_4.z,Pt_4.EdgeColor,Pt_4.DrawColor,SeeFlag,InterFlag,
Type,Pt_4.Edge);
                        m_All_Pt_Array.InsertAt(j0 + 3,Pt_Tmp);
                        flag = 1;
                        break;              //插入完毕,循环下一个
                    }
                    else if(z > Pt_1.z){
//Pt_1 - Pt_2 可见,Pt_3 插入 Pt_1 之前,Pt_4 插入 Pt_1 之后,不可见,Pt_4 缩短插在 Pt_1 之前
                        m_All_Pt_Array.InsertAt(j0,Pt_3);     //Pt_3 插入 Pt_1 之前
                        t = ((double)(Pt_1.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                        z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                        SeeFlag = 1;        //可见,构成点对
                        InterFlag = 0;      //非交点
                        Type = 0;           //普通点
CreateNewPt(Pt_Tmp,Pt_1.x,Pt_1.y,z,Pt_4.EdgeColor,Pt_4.DrawColor,SeeFlag,InterFlag,Type,
Pt_1.Edge);
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_Tmp);     //插在 Pt_1 之前
                        Pt_4.SeeFlag = 0;
                m_All_Pt_Array.InsertAt(j0 + 3,Pt_4);     //Pt_4 不可见,插入 Pt_1 之后
                        flag = 1;
                        break;
                    }
                }
            }
            else if(Pt_3.x >= Pt_1.x&&Pt_4.x <= Pt_2.x){
                //判断 Pt_1 - Pt_2 在 Pt_3 重影点的 z 值
                t = ((double)(Pt_3.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                if(z < Pt_3.z){
                    //Pt_3、Pt_4 不可见,加入为不可见
                    Pt_3.SeeFlag = 0;
                    m_All_Pt_Array.InsertAt(j0 + 1,Pt_3);
                    Pt_4.SeeFlag = 0;
                    m_All_Pt_Array.InsertAt(j0 + 2,Pt_4);
                    flag = 1;
                    break;
                }
                else if(z > Pt_3.z){
                    //Pt_3,Pt_4 可见,加入,将 Pt_1 - Pt_2 分成两段,分别加入端点
                    Pt_Tmp = Pt_1;
                    Pt_Tmp.x = Pt_3.x;
                    Pt_Tmp.z = z;
                    Pt_Tmp.Edge[0] = Pt_3.Edge[0];
                    Pt_Tmp.Edge[1] = Pt_3.Edge[1];
                    if(Pt_3.x == Pt_1.x){
                        Pt_Tmp.SeeFlag = 0;
```

```
                        m_All_Pt_Array.RemoveAt(j0);
                        m_All_Pt_Array.InsertAt(j0,Pt_Tmp);
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_3);     //插入 Pt_3
                        m_All_Pt_Array.InsertAt(j0 + 2,Pt_4);     //插在 Pt_3 之后
                        if(Pt_2.x == Pt_4.x){ //Pt_2 被 Pt_4 挡住
                            m_All_Pt_Array.RemoveAt(j1 + 2);
                            Pt_2.SeeFlag = 0;
                            m_All_Pt_Array.InsertAt(j1 + 2,Pt_2);
                            j += 2;
                            flag = 1;
                            break;
                        }
                        else{                  //计算新 z 值
                            t = ((double)(Pt_4.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                            z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                            SeeFlag = 1;        //可见,构成点对
                            InterFlag = 0;      //非交点
                            Type = 0;           //普通点
CreateNewPt(Pt_Tmp,Pt_4.x,Pt_4.y,z,Pt_1.EdgeColor,Pt_1.DrawColor,SeeFlag,InterFlag,Type,
Pt_4.Edge);
                        m_All_Pt_Array.InsertAt(j0 + 3,Pt_Tmp);     //插在 Pt_4 之后
                            j += 3;             //加了三个点
                            flag = 1;
                            break;
                        }
                    }
                    Else{
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_Tmp);     //插在 Pt_1 之后
                        m_All_Pt_Array.InsertAt(j0 + 2,Pt_3);       //插入 Pt_3
                        m_All_Pt_Array.InsertAt(j0 + 3,Pt_4);       //插在 Pt_3 之后
                        if(Pt_2.x == Pt_4.x){//Pt_2 被 Pt_4 挡住
                        m_All_Pt_Array.RemoveAt(j1 + 3);
                            Pt_2.SeeFlag = 0;
                            m_All_Pt_Array.InsertAt(j1 + 3,Pt_2);
                            j += 3;
                            break;
                        }
                        else{
                            //计算新 z 值
                            t = ((double)(Pt_4.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                            z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                            SeeFlag = 1;        //可见,构成点对
                            InterFlag = 0;      //非交点
                            Type = 0;           //普通点
CreateNewPt(Pt_Tmp,Pt_4.x,Pt_4.y,z,Pt_1.EdgeColor,Pt_1.DrawColor,SeeFlag,InterFlag,Type,
Pt_4.Edge);
                        m_All_Pt_Array.InsertAt(j0 + 4,Pt_Tmp);     //插在 Pt_4 之后
                            j += 4;             //加了四个点
                            flag = 1;
                            break;
                        }
```

```
            }
        }
        else{ //(z == Pt_3.z)
            //判断 Pt_1 - Pt_2 在 Pt_4 重影点的 z 值
            t = ((double)(Pt_4.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
            z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
            if(z < Pt_4.z){
                //Pt_3、Pt_4 不可见,加入为不可见
                Pt_3.SeeFlag = 0;
                m_All_Pt_Array.InsertAt(j0 + 1,Pt_3);
                Pt_4.SeeFlag = 0;
                m_All_Pt_Array.InsertAt(j0 + 2,Pt_4);
                flag = 1;
                j += 2;
                break;
            }
            else if(z > Pt_4.z){
            //Pt_3、Pt_4 可见,加入,将 Pt_1 - Pt_2 分成两段,分别加入端点
                Pt_Tmp = Pt_1;
                Pt_Tmp.x = Pt_3.x;
                Pt_Tmp.z = z;
                Pt_Tmp.Edge[0] = Pt_3.Edge[0];
                Pt_Tmp.Edge[1] = Pt_3.Edge[1];
                if(Pt_3.x == Pt_1.x){
                    Pt_Tmp.SeeFlag = Pt_3.SeeFlag;
                    m_All_Pt_Array.RemoveAt(j0);
                    m_All_Pt_Array.InsertAt(j0,Pt_Tmp);
                    m_All_Pt_Array.InsertAt(j0 + 1,Pt_3);     //插入 Pt_3
                    m_All_Pt_Array.InsertAt(j0 + 2,Pt_4);     //插在 Pt_3 之后
                    if(Pt_2.x == Pt_4.x){//Pt_2 被 Pt_4 挡住
                    m_All_Pt_Array.RemoveAt(j1 + 2);
                        Pt_2.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j1 + 2,Pt_2);
                        j += 2;
                        flag = 1;
                        break;
                    }
                    else{
                        //计算新 z 值
                        t = ((double)(Pt_4.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                        z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                        SeeFlag = 1;          //可见,构成点对
                        InterFlag = 0;        //非交点
                        Type = 0;             //普通点
CreateNewPt(Pt_Tmp,Pt_4.x,Pt_4.y,z,Pt_1.EdgeColor,Pt_1.DrawColor,SeeFlag,InterFlag,Type,
Pt_4.Edge);
                m_All_Pt_Array.InsertAt(j0 + 4,Pt_Tmp);     //插在 Pt_4 之后
                        j += 3;               //加了四个点
                        flag = 1;
                        break;
                    }
```

```
                    }
                    else{
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_Tmp);  //插在 Pt_1 后
                        m_All_Pt_Array.InsertAt(j0 + 2,Pt_3);    //插入 Pt_3
                        m_All_Pt_Array.InsertAt(j0 + 3,Pt_4);    //插在 Pt_3 之后
                        if(Pt_2.x == Pt_4.x){ //Pt_2 被 Pt_4 挡住
                            m_All_Pt_Array.RemoveAt(j1 + 3);
                            Pt_2.SeeFlag = 0;
                            m_All_Pt_Array.InsertAt(j1 + 3,Pt_2);
                            j += 3;
                            break;
                        }
                        else{
                            //计算新 z 值
                            t = ((double)(Pt_4.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                            z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                            SeeFlag = 1;                          //可见,构成点对
                            InterFlag = 0;                        //非交点
                            Type = 0;                             //普通点
CreateNewPt(Pt_Tmp,Pt_4.x,Pt_4.y,z,Pt_1.EdgeColor,Pt_1.DrawColor,SeeFlag,InterFlag,Type,
Pt_4.Edge);
                    m_All_Pt_Array.InsertAt(j0 + 4,Pt_Tmp);       //插在 Pt_4 之后
                            j += 4;                               //加了四个点
                            flag = 1;
                            break;
                        }
                    }
                }
            }
        }
        else if(Pt_3.x >= Pt_1.x&&Pt_3.x < Pt_2.x&&Pt_4.x > Pt_2.x){
            //判断 Pt_1 - Pt_2 在 Pt_3 重影点的 z 值
            t = ((double)(Pt_3.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
            z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
            if(z < Pt_3.z){
                //Pt_3 不可见,加入 Pt_1 之后,产生新的 Pt_3 点,继续判断
                Pt_3.SeeFlag = 0;
                m_All_Pt_Array.InsertAt(j0 + 1,Pt_3);
                //计算新 z 值
                t = ((double)(Pt_2.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                SeeFlag = 1;               //可见,构成点对
                InterFlag = 0;             //非交点
                Type = 0;                  //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_3.EdgeColor,Pt_3.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);
                j++;
                continue;
            }
            else if(z > Pt_3.z){
                //Pt_3 - Pt_4 可见
```

```
                    if(Pt_1.x == Pt_3.x){
                        //Pt_1、Pt_2 都不可见,不删除,设置为不可见
                        m_All_Pt_Array.RemoveAt(j0);
                        Pt_1.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j0,Pt_1);
                        m_All_Pt_Array.RemoveAt(j1);
                        Pt_2.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j1,Pt_2);
                        continue;
                    }
                    else{
                        //Pt_2 不可见,缩短 Pt_2 到 Pt_3.x
                        m_All_Pt_Array.RemoveAt(j1);
                        Pt_2.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j1,Pt_2);
                        //计算新 z 值,缩短 Pt_2 到 Pt_3.x
                        t = ((double)(Pt_3.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                        z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                        SeeFlag = 1;            //可见,构成点对
                        InterFlag = 0;          //非交点
                        Type = 0;               //普通点
CreateNewPt(Pt_2,Pt_3.x,Pt_3.y,z,Pt_2.EdgeColor,Pt_2.DrawColor,SeeFlag,InterFlag,Type,Pt_
3.Edge);
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_2);
                        j++;
                        continue;               //继续判断
                    }
                }
                else {//(z == Pt_3.z)
                    //判断 Pt_3 - Pt - 4 在 Pt_2 的重影点
                    t = ((double)(Pt_2.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                    z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                    if(z < Pt_2.z){
                        //Pt_3 - Pt_4 可见,则缩短 Pt_2 到 Pt_3.x
                        m_All_Pt_Array.RemoveAt(j1);
                        Pt_2.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j1,Pt_2);
                        //计算新 z 值,则缩短 Pt_2 到 Pt_3.x
                        t = ((double)(Pt_3.x - Pt_1.x))/(Pt_2.x - Pt_1.x);
                        z = (Pt_2.z - Pt_1.z) * t + Pt_1.z;
                        SeeFlag = 1;            //可见,构成点对
                        InterFlag = 0;          //非交点
                        Type = 0;               //普通点
CreateNewPt(Pt_2,Pt_3.x,Pt_3.y,z,Pt_2.EdgeColor,Pt_2.DrawColor,SeeFlag,InterFlag,Type,Pt_
3.Edge);
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_2);
m_All_Pt_Array.InsertAt(j0 + 2,Pt_3);        //Pt_3 加入 Pt_2 之后,并产生新 Pt_3,在 Pt_2 处
                        t = ((double)(Pt_2.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                        z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                        SeeFlag = 0;            //不可见,构成点对
                        InterFlag = 0;          //非交点
```

```
                        Type = 0;               //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_3.EdgeColor,Pt_3.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);
                        j++;
                        j++;
                        continue;               //继续判断
                    }
                    else{
                    //Pt_3 不可见,加入 Pt_1 之后,产生新的 Pt_3 点,继续判断
                        Pt_3.SeeFlag = 0;
                        m_All_Pt_Array.InsertAt(j0 + 1,Pt_3);
                        //计算新 z 值
                        t = ((double)(Pt_2.x - Pt_3.x))/(Pt_4.x - Pt_3.x);
                        z = (Pt_4.z - Pt_3.z) * t + Pt_3.z;
                        SeeFlag = 0;            //不可见,构成点对
                        InterFlag = 0;          //非交点
                        Type = 0;               //普通点
CreateNewPt(Pt_3,Pt_2.x,Pt_2.y,z,Pt_3.EdgeColor,Pt_3.DrawColor,SeeFlag,InterFlag,Type,Pt_
2.Edge);
                        j++;
                        continue;
                    }
                }
            }
        }
        if(flag == 0){
            //Pt_3、Pt_4 未处理,则加入最后
            m_All_Pt_Array.Add(Pt_3);
            m_All_Pt_Array.Add(Pt_4);
        }
    }
    return;
}
```

其中,构造新点的函数代码为:

```
void CreateNewPt(CPixelPt& PixelPt, int& x, int& y, double& z, COLORREF& EdgeColor, COLORREF&
DrawColor, int& SeeFlag, int& InterFlag, int& Type, CEdge * Edge){
    PixelPt.x = x;
    PixelPt.y = y;
    PixelPt.z = z;
    PixelPt.EdgeColor = EdgeColor;
    PixelPt.DrawColor = DrawColor;
    PixelPt.Type = Type;
    PixelPt.InterFlag = InterFlag;
    PixelPt.SeeFlag = SeeFlag;
    PixelPt.Edge[0] = Edge[0];
    PixelPt.Edge[1] = Edge[1];
}
```

判断相邻扫描线的交点中同一边上点是否需连线的函数为：

```
void LinkPixelPt(CDC * pDC,CPixelPt& PixelPt0,int j,CArray<CPixelPt,CPixelPt>& m_All_Pt_
Array,CArray<CPixelPt,CPixelPt>& m_All_Pt_Array_Old){
    //从上一条扫描线的交点集中,找到点对应边所在的像素点,判断 x 之间的距离是否过大,如果
过大,则扫描补偿中间点
    CPixelPt PixelPt_0,PixelPt_1,Pt0,Pt1;
    CPoint startPoint,endPoint;
    int flag,YesFlag;
    int iStep = 0;                              //需要判断几次连线 PixelPt0.Type;
                                                //如是顶点,需判断两次
    if(PixelPt0.Type == 0) return;              //非边界点,不连线
    int i,k;
    for(i = 0;i < m_All_Pt_Array_Old.GetSize();i++){
        k = i;
        PixelPt_0 = m_All_Pt_Array_Old.GetAt(i);
        if(PixelPt_0.InterFlag == 0)continue;   //不是交点
        YesFlag = 0;
if(PixelPt0.Edge[0] == PixelPt_0.Edge[0]||PixelPt0.Edge[0] == PixelPt_0.Edge[1]||PixelPt0.
Edge[1] == PixelPt_0.Edge[0]||PixelPt0.Edge[1] == PixelPt_0.Edge[1]){
            if(abs(PixelPt_0.x - PixelPt0.x)> 2)
            {                                   //需插值
                if(PixelPt_0.SeeFlag == 1&&PixelPt0.SeeFlag == 1){    //两个点均可见
                    Pt0 = PixelPt0;
                    Pt1 = PixelPt_0;
                    YesFlag = 1;
                }
else if(PixelPt_0.SeeFlag == 0&&PixelPt0.SeeFlag == 1){    //上一个扫描线的点不可见
        if(PixelPt_0.x < PixelPt0.x){           //找到上一个扫描线上 PixelPt0 前的一个可见点
                    k++;
                    for( ;k < m_All_Pt_Array_Old.GetSize();k++){
                        PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
                        if(PixelPt_1.x > PixelPt0.x){
                            k--;
                            for(;k > 0;k--){
                                PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
if(PixelPt_1.x < PixelPt0.x&&PixelPt_1.SeeFlag == 1&&PixelPt_1.x > PixelPt_0.x){
                                    Pt0 = PixelPt0;
                                    Pt1 = PixelPt_1;
                                    YesFlag = 1;
                                    break;
                                }
                                else
                                    continue;
                            }
                            break;
                        }
                    }
                }
                else{    //找到上一个扫描线上 PixelPt0 后的一个可见点,连线
                    k--;
```

```
                for( ;k>0;k--){
                    PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
                    if(PixelPt_1.x<PixelPt0.x){
                        k++;
                        for(;k<m_All_Pt_Array.GetSize();k++){
                            PixelPt_1 = m_All_Pt_Array.GetAt(k);
if(PixelPt_1.x>PixelPt0.x&&PixelPt_1.SeeFlag==1&&PixelPt_1.x<PixelPt_0.x){
                                Pt0 = PixelPt0;
                                Pt1 = PixelPt_1;
                                YesFlag = 1;
                                break;
                            }
                            else
                                continue;
                        }
                        break;
                    }
                }
            }
        }
    else if(PixelPt_0.SeeFlag==1&&PixelPt0.SeeFlag==0){    //当前扫描线的点不可见
            if(PixelPt_0.x<PixelPt0.x){
                for(k=j-1; k>=0;k--){
                    PixelPt_1 = m_All_Pt_Array.GetAt(k);
                    if(PixelPt_0.x>PixelPt_1.x){
                        k++;
                        for(;k<m_All_Pt_Array.GetSize();k++){
                            PixelPt_1 = m_All_Pt_Array.GetAt(k);
if(PixelPt_1.x>PixelPt_0.x&&PixelPt_1.SeeFlag==1&&PixelPt_1.x<PixelPt0.x){
                                Pt0 = PixelPt_0;
                                Pt1 = PixelPt_1;
                                YesFlag = 1;
                                break;
                            }
                            else
                                continue;
                        }
                        break;
                    }
                }
            }
            else{
                for(k=j+1; k<m_All_Pt_Array.GetSize();k++){
                    PixelPt_1 = m_All_Pt_Array.GetAt(k);
                    if(PixelPt_1.x>PixelPt_0.x){
                        k--;
                        for(;k>0;k--){
                            PixelPt_1 = m_All_Pt_Array.GetAt(k);
if(PixelPt_1.x<PixelPt_0.x&&PixelPt_1.SeeFlag==1&&PixelPt_1.x>PixelPt0.x){
                                Pt0 = PixelPt_0;
                                Pt1 = PixelPt_1;
```

```
                        YesFlag = 1;
                        break;
                    }
                    else
                        continue;
                }
                break;
            }
        }
    }
}
else{//两个扫描线上的点都不可见,分别找最近的可见点作参考
    if(PixelPt_0.x < PixelPt0.x){
        k++;
        for( ;k < m_All_Pt_Array_Old.GetSize();k++){
            PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
            if(PixelPt_1.SeeFlag == 1){
                PixelPt_0 = PixelPt_1;
                break;
            }
        }
        for( -- j; j >= 0;j -- ){
            PixelPt_1 = m_All_Pt_Array.GetAt(j);
            if(PixelPt_1.SeeFlag == 1){
                PixelPt0 = PixelPt_1;
                break;
            }
        }
        if(PixelPt_0.x > PixelPt0.x)
            break;
        else{//继续寻找两线之间的最近可见点
            flag = 0;
            k++;
            for( ;k < m_All_Pt_Array_Old.GetSize();k++){
                PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
        if(PixelPt_1.SeeFlag == 1&&PixelPt_1.x < PixelPt0.x){
                    PixelPt_0 = PixelPt_1;
                    for( -- j; j >= 0;j -- ){
                        PixelPt_1 = m_All_Pt_Array.GetAt(j);
            if(PixelPt_1.SeeFlag == 1&&PixelPt_0.x < PixelPt_1.x){
                            PixelPt0 = PixelPt_1;
                            break;
                        }
                        else{    //已找到最近的两个点
                            flag = 1;
                            break;
                        }
                    }
                }
                else{    //已找到最近的两个点
                    flag = 1;
```

```
                    }
                    if(flag == 1){
                        Pt0 = PixelPt_0;
                        Pt1 = PixelPt0;
                        YesFlag = 1;
                        break;
                    }
                }
            }
        }
        else{//分别找最近的可见点
            for(++j; j < m_All_Pt_Array.GetSize();j++){
                PixelPt_1 = m_All_Pt_Array.GetAt(j);
                if(PixelPt_1.SeeFlag == 1){
                    PixelPt0 = PixelPt_1;
                    break;
                }
            }
            for( -- k;k > 0;k -- ){
                PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
                if(PixelPt_1.SeeFlag == 1){
                    PixelPt_0 = PixelPt_1;
                    break;
                }
            }
            if(PixelPt0.x > PixelPt_0.x)
                break;
            else{//继续寻找两线之间的最近可见点
                flag = 0;
                for( -- k;k > 0;k -- ){
                    PixelPt_1 = m_All_Pt_Array_Old.GetAt(k);
            if(PixelPt_1.SeeFlag == 1&&PixelPt_1.x > PixelPt0.x){
                        PixelPt_0 = PixelPt_1;
                        for(++j; j < m_All_Pt_Array.GetSize();j++){
                            PixelPt_1 = m_All_Pt_Array.GetAt(j);
if(PixelPt_1.SeeFlag == 1&&PixelPt_0.x > PixelPt_1.x){
                                PixelPt0 = PixelPt_1;
                                break;
                            }
                            else{    //已找到最近的两个点
                                flag = 1;
                                break;
                            }
                        }
                    }
                    else{    //已找到最近的两个点
                        flag = 1;
                    }
                    if(flag == 1){
                        Pt0 = PixelPt_0;
                        Pt1 = PixelPt0;
                        YesFlag = 1;
```

```
                            break;
                        }
                    }
                }
            }
        }
        //连线,判断是否继续
        if(YesFlag == 1){
            startPoint.x = Pt0.x;
            startPoint.y = Pt0.y;
            endPoint.x = Pt1.x;
            endPoint.y = Pt1.y;
        MIDPOINT_Line(pDC,startPoint,endPoint,PixelPt0.EdgeColor);
        }
        if(PixelPt0.Type >= 2)
                continue;
            else
                break;
    }
  }
 }
}
```

在程序中为了实现一般多面体的消隐，在OnDraw()函数中对于拉伸实体的显示部分作如下修改：

```
        if(m_HideFlag == false){
            for(int i = 0;i < num_Body;i++){
if((m_Picker.picktype == pick_body&&m_Body[i] == m_Picker.m_Body_Stretch) == false)
    DrawBody(pDC,m_Body[i],m_DrawColor,m_Matrix_V,m_Body,num_Body,m_LineWidth,m_lineType,
m_HideFlag);
            }
        }
        else{
    DrawTrueBody(pDC,m_Picker,HIGHLIGHTCOLOR,m_Matrix_V,m_Body,m_DrawColor,num_Body,m_
RenderFlag,m_LightVector,m_dblEnvironment,m_dblDefuse,m_dblMirror,m_iNs,m_dblFatt);
        }
```

图6.3-9所示为上述消隐算法对任意一个线框拉伸实体的消隐效果及对比，从结果看，上述扫描线消隐算法是有效可行的。

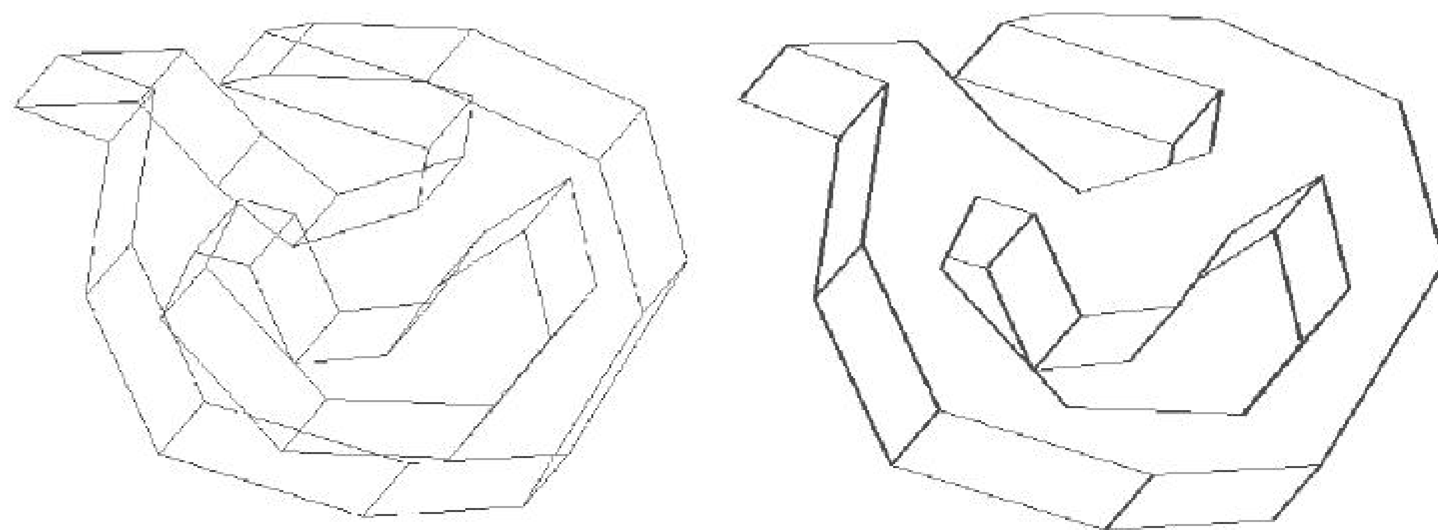


图6.3-9 消隐前后效果对比

第 7 章 真实感图形绘制

为了使计算机所绘制出的图形能更真实地再现物体,并与真实世界中的物体更接近,除了对图形进行消隐外,还需显示图形表面的颜色、亮度、材质、纹理等特征,此时的图形称为真实感图形。在真实世界中,只有在有光的环境下才能看到物体;计算机绘制真实感图形时,同样也需要根据假定的光照条件和景物外观因素,依据一定的光照模型,计算可见面投射到观察者眼中的光强度大小,并将它转换成适合图形设备的颜色值,生成投影画面上每一个像素的光强度,从而使观察者产生身临其境的感觉,这就是计算机绘制真实感图形的思路。真实感图形绘制是计算机图形学的一个重要组成部分,真实感图形绘制技术有重要的实用价值,在工业设计和实物模型制作方面,采用计算机真实感图形绘制技术,可方便地在屏幕上显示产品各种角度的真实感视图,并在屏幕上直接对外形进行交互式的修改,减少大量的人力物力成本;真实感图形绘制在计算机仿真技术、模拟自然景物、科学计算可视化模拟(如分子结构、星体运行等微观和宏观世界许多看不到的物理现象)和处理海量数据、计算机动画、虚拟现实以及飞行训练、战斗模拟、医学、影视广告等领域都有广阔的应用前景。

计算场景中可见面的颜色,严格地说,是根据基于光学物理的光照明模型计算可见面投影到观察者眼中的光亮度大小和色彩组成,并将它转换成适合图形设备的颜色值,从而确定投影画面上每一像素的颜色,最终生成图形。所以,本章首先对相关的光学原理和颜色等概念进行简单介绍。

7.1 相关物理知识

7.1.1 基本光学原理

光是真实世界能看到物体的基本条件,黑暗中是无法识别物体的,而光是由光源发出的,光源的类型、光源的位置以及光源的强度大小对物体的显示效果有直接的影响,因此,光源是计算机绘制真实感图形的一个重要影响因素。

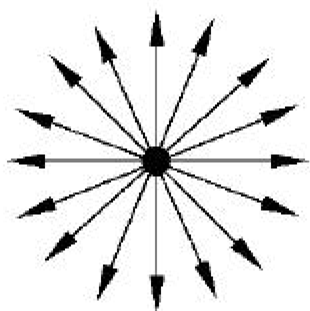


图 7.1-1 点光源

从几何特性上来讲,光源的种类有点光源、平行光源、线光源、面光源、体光源以及环境光等。点光源可以假定光线是从一个点向四周均匀发射,如图 7.1-1 所示,灯泡或者蜡烛的光就可以认为是点光源

发出的。多个连续的点光源合在一条线上就构成了线光源，如荧光灯管。当发出的光线是平行照射时，此光源是平行光源，例如，太阳光就是平行光。同理，面光源、体光源以及环境光都是指产生光线的光源特征。

光的强弱可以用光强表示，光强也表示光能的大小，一般认为点光源向四周发出的光强相同，但会衰减，公式为 $I_a=\frac{I_p}{d^2}$，其中 d 为距离。认为平行光和环境光的光强不衰减，而线光源、面光源和体光源的光强衰减比较复杂，且只在特定环境下存在，故在图形学中常用的光源是点光源、平行光源和环境光。

光的一个非常重要的物理特性是光的波长即光谱，不同波长的光会产生不同的颜色。在人们的眼中，波长 630nm 的光是红色的，波长 530nm 的光是绿色的，波长 450nm 的光是蓝色的。这三种波长的光对视网膜的锥状细胞的刺激最强，视觉理论把红、绿、蓝这三种颜色作为颜色基色，又称 RGB 三原色。不同强度的几种基色加在一起可以形成另一种颜色，如图 7.1-2 所示。

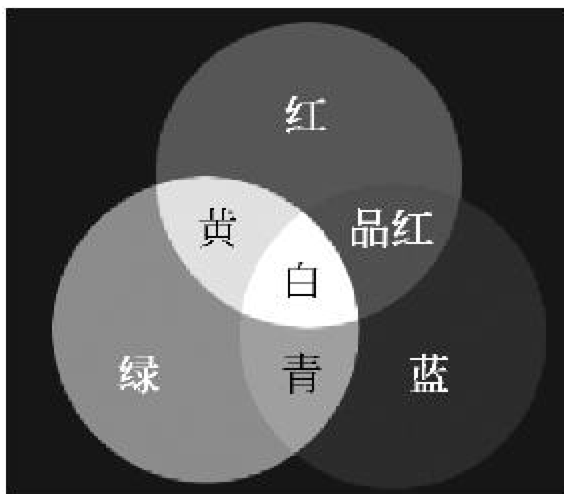


图 7.1-2　RGB 颜色模型

7.1.2　颜色与光的关系

虽然物体由于材质和表面处理的不同具有不同的客观颜色，但是，我们看到物体表面的颜色效果却取决于物体表面的反射光和透射光的光谱分布，以及物体表面对入射光中不同波长的光的反射程度，我们的视觉系统——眼睛是通过可见光对视网膜的锥状细胞的刺激来感受颜色的，所以，我们看到的物体颜色是一个主观感觉。

我们之所以能看到各种颜色，是由于光线照射到着色的物体表面时，物体有选择地吸收了一部分波长的光线而反射出另一部分波长的光线。反射出来的光线进入人的眼睛，人们所看到的颜色就是物体反射出来的那些波长的颜色。大多数物体对白光中各种不同的光波具有不同的吸收率，这些物体称为选择性吸收体。当白光照射一块蓝玻璃时，蓝玻璃吸收了红光、绿光，透过了蓝光，呈蓝色。一块蓝布是因为吸收了红光、绿光，只吸收了很少蓝光，反射了大部分蓝光而显蓝色。所以，绝大多数非发光体的颜色取决于经物体选择性吸收以后所反射或透射出来的光线的光谱成分，人眼所看到的非发光体的颜色就是该物体所不吸收或吸收较少的颜色。一般情况下，当物体表面是某种颜色时，那么对该种颜色波长的光吸收较少，而反射或者透射较多。

当物体对某种颜色吸收较多时，那么对该颜色的反射就较少，也即反射率较低。当我们假定入射光、反射光和透射光都仅仅由红(R)、绿(G)、蓝(B)三种基色组成时，假设某物体表面对 RGB 三个颜色基色的反射率分别是(1.0,0.5,0)，则它能反射全部红光，反射一半绿光，不反射蓝光。现在，假定有一个光源，它的 RGB 值为(L_R,L_G,L_B)，物体表面的反射率是(M_R,M_G,M_B)，当该光源照射到物体上时，在不考虑其他反射效果的前提下，眼睛看到的反射光的颜色是(L_RM_R,L_GM_G,L_BM_B)。一般情况下，光源都是白光即 RGB(255,255,255)，假设物体表面颜色是红色，即 RGB(255,0,0)，则物体表面对白光的反射率为(1.0,0,0)，眼睛看到的这个光源的颜色为 RGB(255,0,0)。

计算机图形中常用的颜色模型有 RGB 颜色模型、CMY 颜色模型、HSV 颜色模型等。

CMY 颜色模型为以红、绿、蓝的补色青(cyan)、平红(magenta)、黄(yellow)为原色构成的模型,常用于从白光中滤去某种颜色,又称减性原色系统,在印刷行业中,基本上都是使用这种颜色模型。RGB 和 CMY 颜色模型都是面向硬件的,它与直观的颜色概念(如色调、色饱和度和明度值)没有直接关系,而 HSV(hue saturation value)颜色模型是面向用户的。HSV 颜色模型对应于画家的配色方法,画家用改变色泽和色深的方法从纯色获得不同色调的颜色,例如,在纯色中加入白色以改变色泽,加入黑色以改变色深,同时加入不同比例的白色、黑色即可得到不同色调的颜色。本书以 RGB 颜色模型来建立解决物体表面颜色显示的光照模型。

7.2 光照模型

7.2.1 简单光照模型

所谓光照模型(illumination model),指的是对光照射到物体表面所产生的反射、折射现象的模拟,就是根据光学物理的有关定律,计算物体表面各点投射到观察者眼中的光线的光亮度和色彩组成的数学公式。它也称明暗模型,主要用于物体表面某点处的光强度计算。计算机图形学的光照模型的作用就是解决如何计算物体表面的红、绿、蓝三种基色的颜色值的问题。

物体的表面性质和自然界中的真实光线对物体表面亮度的影响很难精确地模拟,只能逼近实际条件,与实际条件越接近,所得的光照模型就越复杂,计算量也就越大,逼真性越强。根据计算机模拟物体表面对光的大小颜色的反映与现实接近的程度不同,形成不同的光照模型。

(1) 简单光照模型。一般情况下,只需考虑光源的漫反射和镜面反射,此时所得的光照模型称为局部光照模型。

(2) 复杂光照模型。复杂的光照模型除了考虑反射光外,还要考虑其他一些因素:周围环境的光对物体表面的影响,物体的透明度,阴影的处理,物体表面细节的处理,光源的位置和个数等。这类光照模型称为整体光照模型,它使绘制的图形更接近自然景物。

(3) 局部模型。它是一个经验模型,但能在较短时间内获得具有一定真实感的图形,能较好地模拟光照效果和镜面高光,且计算简单,所涉及的参数量易于获得,因此在实际中得到了广泛的应用和推广,是目前三维图形真实感处理技术所采用的主要方法。简单光照模型中只考虑反射光的作用。反射光由环境光、漫反射光和镜面反射光三部分组成。

1. 环境光

环境光(background light)可看作邻近各物体所产生的光多次反射和散射,最终达到平衡时的一种光,产生的作用是虽然物体没有受到光源的直接照射,但其表面仍有一定的亮度。环境光的特点是照射在物体上的光来自周围各个方向,又均匀地向各个方向反射,如图 7.2-1 所示。

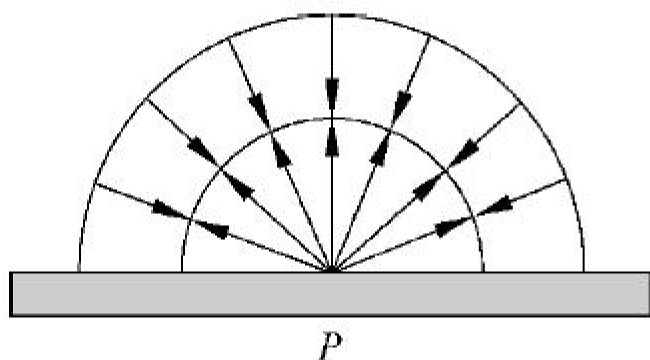


图 7.2-1 环境光

如果用 I_a 表示环境光的入射强度的大小，则物体表面上一点对环境光的反射强度 I_e 可表示为

$$I_e = I_a k_a, \quad 0 \leqslant k_a \leqslant 1$$

式中，k_a 为由物体材质决定的表面对环境光的反射系数，即反射率。同一环境下的环境光是恒定不变的，对任何物体的表面的强度和亮度都相同，这样，同一个物体的所有可见表面在只有环境光照射下的明暗度是一样的，区分不出物体哪些表面明亮，哪些表面暗淡，因此不能产生真实感图形的效果。

由于物体表面对光的吸收和反射作用，如果假设物体对与表面颜色相同的光完全反射，对于颜色不相同的光完全吸收，则通过环境光反射看到物体的颜色，即为物体真实的材质颜色。

2. 漫反射光

漫反射光(diffuse light)可以看作在点光源的照射下，光被物体表面吸收后，然后重新反射出来。漫反射光的特点是光源来自一个方向，反射光均匀地射向各个方向。

如图 7.2-2 所示，设物体表面在 P 点的法向量为 $\boldsymbol{N}$，从 P 点指向光源的向量为 $\boldsymbol{L}$；$\boldsymbol{N}$ 与 $\boldsymbol{L}$ 的夹角为 θ；若 $\boldsymbol{N}$ 与 $\boldsymbol{L}$ 的夹角小于 0°或大于 90°，则光线被物体自身遮挡而照射不到 P 点。由 Lambert 余弦定理可得点 P 处漫反射光的强度为

$$I_d = I_p K_d \cos\theta, \quad 0 \leqslant \theta \leqslant 90°$$

式中，I_p 为入射光的强度，K_d 为漫反射系数。

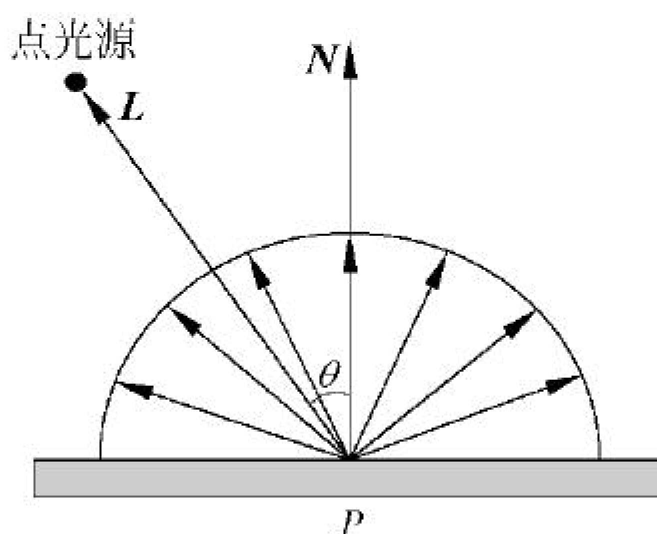


图 7.2-2　漫反射光

如果 $\boldsymbol{N}$ 和 $\boldsymbol{L}$ 是规格化的，对于单位矢量，$\cos\theta = \boldsymbol{N} \cdot \boldsymbol{L}$，则 Lambert 漫反射模型可以写为

$$I_d = I_p K_d (\boldsymbol{L} \cdot \boldsymbol{N})$$

3. 镜面反射光

如果光照射到光滑的表面，则会发生镜面反射，镜面反射的特点是在光滑表面会产生一块高光的区域。利用镜面反射可以很好地模拟光滑物体在光照下表面产生高亮的现象。

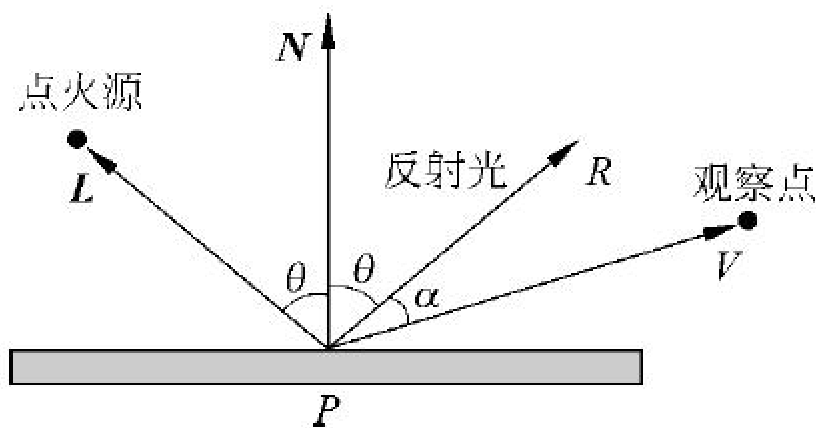


图 7.2-3　镜面反射光

镜面反射遵循光的反射定律。如图 7.2-3 所示，反射光与入射光位于表面法向两侧，对于理想反射表面(如镜面)，入射角等于反射角，观察者只能在表面法向的反射方向一侧才能看到反射光。

镜面反射情况由经验模型 Phong 模型给出：

$$I_s = I_p K_s \cos^{n_s}\alpha, \quad 0° \leqslant \alpha \leqslant 90°$$

式中，I_p 为入射光的强度；K_s 为物体表面某点的高亮光系数，即镜面反射率；α 为 $\boldsymbol{R}$ 与 $\boldsymbol{V}$ 的夹角；n_s 为物体表面的镜面反射指数，取值范围为 1～2000，反映物体表面的光滑程度，表面越光滑，n_s 越大。

假定所有的矢量均为单位矢量，令 $\boldsymbol{H}=(\boldsymbol{L}+\boldsymbol{V})/2$，即 $\boldsymbol{H}$ 为 $\boldsymbol{L}$ 和 $\boldsymbol{V}$ 的角平分矢量，则 $\boldsymbol{N}$ 与 $\boldsymbol{H}$ 的夹角是 α，因此，$\cos\alpha = \boldsymbol{V} \cdot \boldsymbol{R} = \boldsymbol{N} \cdot \boldsymbol{H}$，则镜面反射光模型改写为

$$I_s = I_p K_s (\boldsymbol{N} \cdot \boldsymbol{H})^{n_s}$$

从视点观察到物体上任一点 P 处的光强度 I 应为环境光反射光强度 I_e、漫反射光强度 I_d 以及镜面反射光的光强度 I_s 的总和，此即 Phone 光照模型：

$$I = I_e + I_d + I_s = I_a K_a + I_p K_d (\boldsymbol{L} \cdot \boldsymbol{N}) + I_p K_s (\boldsymbol{N} \cdot \boldsymbol{H})^{n_s}$$

4. 多光源的情况

当有多个光源时，例如有 m 个光源照射，在计算物体可见表面某点的光强时，需要逐个累加每个光源的漫反射和镜面反射的光强贡献，这时 Phone 光照模型需修改为

$$I = I_a K_a + \sum_{i=1}^{m} [I_{pi} K_{di} (\boldsymbol{L}_i \cdot \boldsymbol{N}) + I_{pi} K_{si} (\boldsymbol{N} \cdot \boldsymbol{H}_i)^{n_s}]$$

5. 光的衰减

点光源在空间发出的光线强度会随光行进距离的增加而衰减，物体某点的入射光强度与该点和光源距离的平方成反比，在计算机图形学中，采用衰减因子来表示光的衰减：

$$f_{\text{att}} = \min\left(1, \frac{1}{c_0 + c_1 d + c_2 d^2}\right)$$

式中，c_0、c_1、c_2 为常数，用于保证衰减因子在 1 之内，以确保总是衰减的。

这样，在考虑了光的衰减后，Phone 光照模型进一步修改为

$$I = I_a K_a + \sum_{i=1}^{m} f_i [I_{pi} K_{di} (\boldsymbol{L}_i \cdot \boldsymbol{N}) + I_{pi} K_{si} (\boldsymbol{N} \cdot \boldsymbol{H}_i)^{n_s}]$$

式中，f_i 为第 i 个光源的衰减因子。

6. Phone 光照模型的 RGB 颜色模型形式

在 RGB 颜色模型中，把入射光强 I 分为三个分量，分别代表 RGB 三基色的光强，通过这些分量的值来调整光源的颜色。同样的，K_a、K_d、K_s 也有三个分量。于是，Phone 光照模型的 RGB 颜色模型形式为

$$I_{\mathrm{R}} = I_{a\mathrm{R}} K_{a\mathrm{R}} + \sum_{i=1}^{m} f_i [I_{pi\mathrm{R}} K_{di\mathrm{R}} (\boldsymbol{L}_i \cdot \boldsymbol{N}) + I_{pi\mathrm{R}} K_{si\mathrm{R}} (\boldsymbol{N} \cdot \boldsymbol{H}_i)^{n_s}]$$

$$I_{\mathrm{G}} = I_{a\mathrm{G}} K_{a\mathrm{G}} + \sum_{i=1}^{m} f_i [I_{pi\mathrm{G}} K_{di\mathrm{G}} (\boldsymbol{L}_i \cdot \boldsymbol{N}) + I_{pi\mathrm{G}} K_{si\mathrm{G}} (\boldsymbol{N} \cdot \boldsymbol{H}_i)^{n_s}]$$

$$I_{\mathrm{B}} = I_{a\mathrm{B}} K_{a\mathrm{B}} + \sum_{i=1}^{m} f_i [I_{pi\mathrm{B}} K_{di\mathrm{B}} (\boldsymbol{L}_i \cdot \boldsymbol{N}) + I_{pi\mathrm{B}} K_{si\mathrm{B}} (\boldsymbol{N} \cdot \boldsymbol{H}_i)^{n_s}]$$

上述的 Phone 光照模型只考虑了环境光、漫反射光以及镜面反射光，是一种简单光照模型，利用上述模型生成的图形真实度已经达到了可以接受的程度。

在程序中实现上面的光照模型时，首先要实现物体的消隐处理。对于潜在可见面，利用 Phone 光照模型的 RGB 颜色模型形式计算物体表面的每一个点的颜色，并在对应像素点用该颜色填充，即得真实感图形。

在第 6 章的消隐技术中，利用扫描线法实现了一般多面体的消隐。扫描线算法不仅可以实现边和面的消隐，也可以通过将扫描线与可见面投影的交点对面内的像素点进行填充，从而获得表面的真实感颜色。

在具体编程实现时，假设只有一个光源，光源是一般的白光，即 RGB(255,255,255)。将第 6 章扫描线算法的相关程序代码进行部分修改，加入计算表面颜色码以及显示的代码。

首先，Phone 光照模型的相关参数通过创建一个非模式对话框 CRenderDlg 进行具体

设置，如图7.2-4所示，非模式对话框的创建方法见本书前面相关章节。在该对话框中，设置是否光照渲染的标识、拾取形体的颜色、光源的位置以及Phone模型的相关参数。

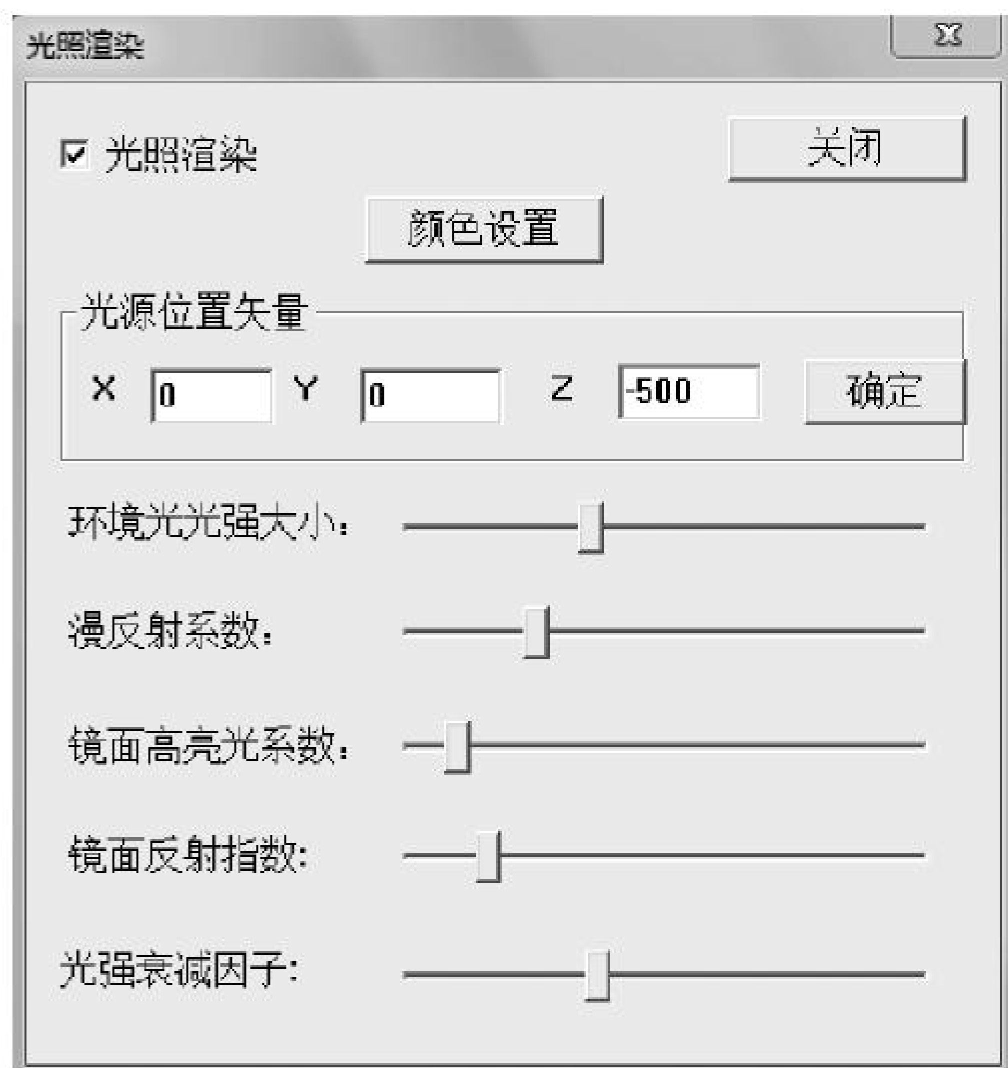


图7.2-4　Phone模型参数设置

其中，设置所拾取形体颜色的代码如下：

```
void CRenderDlg::OnButtonColor() {
    //设置拾取实体的颜色
    CColorDialog dlg;
    int i = 0;
    for(i = 0;i < this -> m_pView -> num_Body;i++){
if((this -> m_pView -> m_Picker.picktype == pick_body&&this -> m_pView -> m_Body[i] == this ->
m_pView -> m_Picker.m_Body_Stretch) == true){
            dlg.m_cc.rgbResult = this -> m_pView -> m_Body[i].color;
            dlg.m_cc.Flags| = CC_RGBINIT|CC_FULLOPEN;
            if(IDOK == dlg.DoModal()){
                this -> m_pView -> m_Body[i].color = dlg.m_cc.rgbResult; //将\dlg.m_cc.
//rgbResult 获取到的颜色对话框中的颜色保存到变量 m_clr 中
            }
            break;
        }
    }
}
```

设置点光源位置矢量的代码如下：

```
void CRenderDlg::OnOK() {
    //设置光源矢量
    UpdateData(TRUE);
    this -> m_pView -> m_LightVector.v_x = this -> m_V_x;
    this -> m_pView -> m_LightVector.v_y = this -> m_V_y;
    this -> m_pView -> m_LightVector.v_z = this -> m_V_z;
    this -> m_pView -> Invalidate();
```

```
    UpdateData(FALSE);
}
```

使用光源位置矢量之前，需要在 BasicClass.h 中声明一个矢量类，类结构如下：

```
class CVector{///矢量类
public:
    double v_x, v_y, v_z, v_s;
    Vector(){
    v_x = 0.0;v_y = 0.0;v_z = 0.0;v_s = 1.0;
    }
    bool operator == (CVector &Vector){
        if(this == &Vector) return true;
        else if(this -> v_x == Vector.v_x&&this -> v_y == Vector.v_y&&this -> v_z == Vector.v_z)
            return true;
        else
            return false;
    }
};
```

在 OnInitDialog()函数中初始化 Phone 模型的相关参数，代码如下：

```
BOOL CRenderDlg::OnInitDialog() {
    CDialog::OnInitDialog();
    m_E.SetRange(0,100);                    //环境光系数范围
    m_E.SetLineSize(5);
    m_D.SetRange(0,100);                    //漫反射系数范围
    m_D.SetLineSize(5);
    m_M.SetRange(0,100);                    //镜面反射系数范围
    m_M.SetLineSize(5);
    m_Ns.SetRange(1,100);                   //镜面反射指数范围
    m_Ns.SetLineSize(5);
    m_F.SetRange(1,100);                    //点光源衰减因子
    m_F.SetLineSize(5);
    return TRUE;
}
```

在滑动控件上滑动时，可以实时改变相关的参数值，并对图形直接重新生成。代码如下：

```
void CRenderDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar){
    UpdateData(TRUE);
    if(pScrollBar -> GetDlgCtrlID() == IDC_SLIDER_E){           //环境光强系数
        m_iEnvironment = ((CSliderCtrl * )pScrollBar) -> GetPos();
        this -> m_pView -> m_dblEnvironment = m_iEnvironment/100.0;
    }
    else if(pScrollBar -> GetDlgCtrlID() == IDC_SLIDER_D){      //漫反射系数
        m_iDefuse = ((CSliderCtrl * )pScrollBar) -> GetPos();
        this -> m_pView -> m_dblDefuse = m_iDefuse/100.0;
    }
    else if(pScrollBar -> GetDlgCtrlID() == IDC_SLIDER_M){      //镜面反射高亮光系数
        m_iMirror = ((CSliderCtrl * )pScrollBar) -> GetPos();
```

```
        this->m_pView->m_dblMirror = m_iMirror/100.0;
    }
    else if(pScrollBar->GetDlgCtrlID() == IDC_SLIDER_NS){    //镜面反射指数
        m_iMirror_ns = ((CSliderCtrl*)pScrollBar)->GetPos();
        this->m_pView->m_iNs = m_iMirror_ns;
    }
    else if(pScrollBar->GetDlgCtrlID() == IDC_SLIDER_F){     //光衰减系数
        m_iFatt = ((CSliderCtrl*)pScrollBar)->GetPos();
        this->m_pView->m_dblFatt = m_iFatt/100.0;
    }
    this->m_pView->Invalidate();
    UpdateData(FALSE);
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

为了实现光照模型，需对绘制消隐的函数 DrawTrueBody()的参数以及内容进行修改，在参数中加入 Phone 光照模型中的相关参数。该函数的参数修改如下：

```
void DrawTrueBody(CDC* pDC,CPicker &m_Picker,COLORREF m_EdgeColor,double m_Matrix_V[][4],
CBody_Stretch Body[], COLORREF &m_DrawColor, int bodyNum, bool& m_RenderFlag, CVector& m_
LightVector,double &m_dblEnvirenment,double& m_dblDefuse,double& m_dblMirror,int& m_iNs,
double& m_dblFatt)
```

其中，新加入的参数有：m_RenderFlag 为渲染显示的标识符；m_LightVector 为光源的位置矢量；m_dblEnvironment 为环境光反射系数；m_dblDefuse 为漫反射系数；m_dblMirror 为镜面反射系数即高亮光系数；m_iNs 为镜面反射指数；m_dblFatt 为衰减因子。

在函数代码中，对于调用计算表面的可见性及占有扫描线的函数 BuildBodySurf()，加入表面的光照颜色的处理内容，这时该函数的参数中也要加入光照模型的相关参数：

```
void BuildBodySurf(CBodySurf& BodySurf,bool& m_RenderFlag,CVector& m_LightVector,double &m_
dblEnvironment,double& m_dblDefuse,double& m_dblMirror,int& m_iNs,double& m_dblFatt)
```

其中，新加入的各参数的意义和 DrawTrueBody()中加入的参数意义相同。

在 BuildBodySurf()函数中，当某表面是潜在可见时，加入光照颜色的计算，代码如下：

```
if(VectorXVectorForZ(pt0_3D,pt1_3D,pt2_3D)<0){          //矢量叉乘 Z<0
        BodySurf.BodySurf_See_Flag = TRUE;              //潜在可见
        (下面为新加入部分)
        if(m_RenderFlag == true){
            //计算渲染
            double m_dblE/*环境光强*/,m_dblD/*漫反射光强*/,m_dblM/*镜面反射
光强*/,m_total/*合计平均光强*/;
            CVector N_Vector,V_Vector,H_Vector;
            //计算环境光强
            m_dblE = m_dblEnvironment;
            //漫反射光强
            //计算表面法矢
            VectorXVector(pt0_3D,pt1_3D,pt2_3D,N_Vector);
            //L.N计算视线向量和表面法矢的点乘
```

```
                double m_LdotN = VectorDotVector(m_LightVector,N_Vector);
                if(m_LdotN < 0)
                    m_LdotN = 0;
                m_dblD = m_dblFatt * m_dblDefuse * m_LdotN;
                //镜面反射
                V_Vector.v_z = 1.0;
                V_Vector.v_x = 0.0;
                V_Vector.v_y = 0.0;
                double
sqrt0 = sqrt(m_LightVector.v_x * m_LightVector.v_x + m_LightVector.v_y * m_LightVector.v_y +
m_LightVector.v_z * m_LightVector.v_z);
                m_LightVector.v_x = m_LightVector.v_x/sqrt0;
                m_LightVector.v_y = m_LightVector.v_y/sqrt0;
                m_LightVector.v_z = m_LightVector.v_z/sqrt0;
                H_Vector.v_x = (m_LightVector.v_x + V_Vector.v_x)/2;
                H_Vector.v_y = (m_LightVector.v_y + V_Vector.v_y)/2;
                H_Vector.v_z = (m_LightVector.v_z + V_Vector.v_z)/2;
                //计算点光源和表面法向量的点乘
                double m_HdotN = VectorDotVector(H_Vector,N_Vector);
                if(m_HdotN < 0)m_HdotN = 0;
                m_dblM = m_dblMirror * pow(m_HdotN,m_iNs);
                m_total = (m_dblE + m_dblD + m_dblM)/3;//假设三种光的权重相同
                int iR = GetRValue(BodySurf.color);//
                int iG = GetGValue(BodySurf.color);
                int iB = GetBValue(BodySurf.color);
                iR = iR * m_total;              //计算红色分量
                iG = iG * m_total;              //计算绿色分量
                iB = iB * m_total;              //计算红色分量
                BodySurf.color = RGB(iR,iG,iB);  //表面内的点设置颜色
            }
        }
```

其中,计算两个向量叉乘和点乘的函数代码为:

```
void VectorXVector(CPoint3D &pt0,CPoint3D &pt1,CPoint3D &pt2,CVector& Vector){
    //计算向量的叉乘
    Vector.v_z = (pt1.x - pt0.x) * (pt2.y - pt1.y) - (pt2.x - pt1.x) * (pt1.y - pt0.y);
    Vector.v_x = (pt1.y - pt0.y) * (pt2.z - pt1.z) - (pt2.y - pt1.y) * (pt1.z - pt0.z);
    Vector.v_y = (pt1.z - pt0.z) * (pt2.x - pt1.x) - (pt2.z - pt1.z) * (pt1.x - pt0.x);
}
double VectorDotVector(CVector& Vector1,CVector& Vector2){
    //计算向量的点乘,并单位化
    return (Vector1.v_x * Vector2.v_x + Vector1.v_y * Vector2.v_y + Vector1.v_z * Vector2.v_
z)/sqrt((Vector1.v_x * Vector1.v_x + Vector1.v_y * Vector1.v_y + Vector1.v_z * Vector1.v_z) +
(Vector2.v_x * Vector2.v_x + Vector2.v_y * Vector2.v_y + Vector2.v_z * Vector2.v_z));
}
```

在 DrawTrueBody()函数中,当一条扫描线与潜在可见面求交点并排序后,如果光照渲染标识符是 TRUE,则对交点对内的像素填充颜色。相关位置的代码及新加入的代码如下:

```
for(yscan = ymin;yscan < = ymax;yscan++){
        m_All_Pt_Array.RemoveAll();
        for(s = 0;s < BodySurf_num;s++){
if(Body_Surf[s].BodySurf_See_Flag == TRUE&&yscan > = Body_Surf[s].y_min_Project&&yscan < =
Body_Surf[s].y_max_Project)
                //扫描该面,获得交点对
                m_OneSurf_Pt_Array.RemoveAll();
                if(Body_Surf[s].PickFlag == 1)
ScanProjectSurf(yscan,Body_Surf[s],m_EdgeColor,m_OneSurf_Pt_Array);
                else
ScanProjectSurf(yscan,Body_Surf[s],m_DrawColor,m_OneSurf_Pt_Array);
                //从 m_OneSurf_Pt_Array 中取出一对,插入 m_All_Pt_Array;
                InsertPtArray(m_All_Pt_Array,m_OneSurf_Pt_Array);
        }
        if(m_RenderFlag == false){
              (消隐部分,前一章代码已列出,不再赘述)
        }
        else{//光照渲染
            for(j = 0;j < m_All_Pt_Array.GetSize();j++)//,j++){
                PixelPt0 = m_All_Pt_Array.GetAt(j);
                if(PixelPt0.SeeFlag == 0)     //不可见,查找下一个可见点
                    continue;
                for(k = j + 1;k < m_All_Pt_Array.GetSize();k++){
                    PixelPt = m_All_Pt_Array.GetAt(k);
                    if(PixelPt.SeeFlag == 0)
                        continue;
                    else{
                        j = k;
                        //从 PixelPt0 填充到 PixelPt
                        for(i = PixelPt0.x;i < PixelPt.x;i++){
                            pDC - > SetPixel(i,PixelPt0.y,PixelPt0.DrawColor);
                        }
                        break;
                    }  }  }
        }
    m_All_Pt_Array_Old.RemoveAll();
    m_All_Pt_Array_Old.Append(m_All_Pt_Array);
    }
```

采用上述方法实现的光照渲染效果如图 7.2-5 所示,该形体也为图 6.3-9 所示的拉伸线框实体实现的光照效果。

7. 马赫带效应

在前面讨论的光照模型中,最后的显示结果依赖于物体表面法向量的计算,使用这种基本数学模型计算物体表面明暗度的过程称为明暗效应处理。当物体表面为曲面时,通常用一组多边形(三角片)来近似模拟曲面的显示状态,那么,一个多边形中所有的点都对应于一个光强度,该多边形面上所有的点都用相同的光强度值来显示,这时,相邻接的多边形会存在光强突变,产生一种称为马赫带效应(Mach band effect)的现象。如图 7.2-6 所示,牛的表面利用多边形(三角片)逼近后,再利用明暗效应处理,这时整个表面的光强不再连续,而是出现图中左侧图的现象,即马赫带效应。

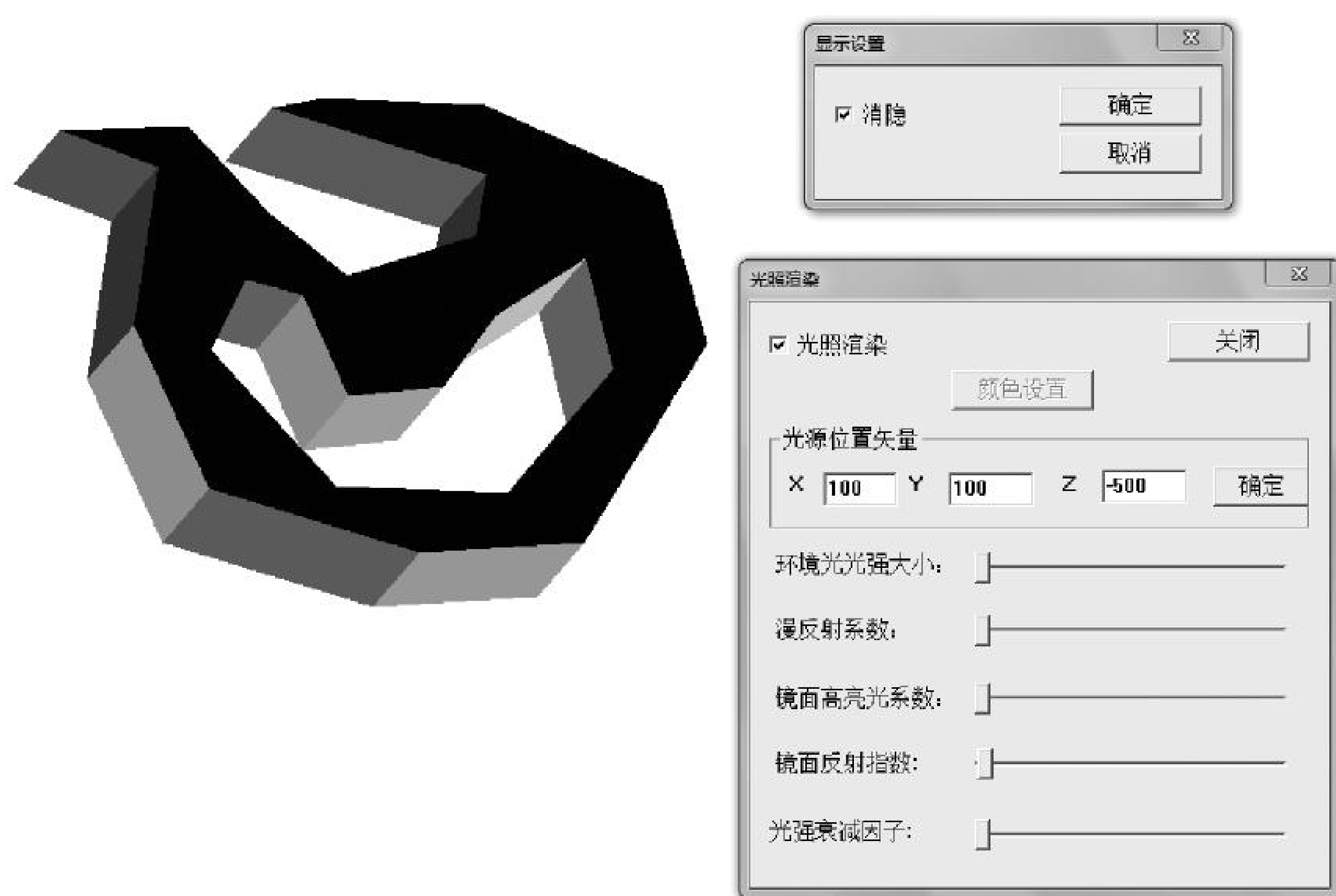


图 7.2-5 Phone 模型光照效果图



图 7.2-6 马赫带效应

出现马赫带效应有下面三个前提条件:

(1) 物体是一个多面体,而不是曲面物体的近似表示;

(2) 所有的光源都离物体足够远,以至于 $\boldsymbol{N}\cdot\boldsymbol{L}$ 和受距离影响的衰减对物体表面近似地成为一个常数;

(3) 视点离物体表面足够远,以至于 $\boldsymbol{V}\cdot\boldsymbol{R}$ 或 $\boldsymbol{N}\cdot\boldsymbol{H}$ 对物体表面成为一个近似不变的常数。

马赫带效应是视觉系统夸大了具有不同常量光强的两个相邻区域之间的光强,使之具有不连续性。

为了保证多边形之间的光滑过渡,使连续的多边形呈现匀称的光强,需要根据某种规则对多边形顶点间的信息进行加权平均,即插值。其基本思想是在每一个多边形的顶点处计算出光强或参数,然后在各个多边形内部进行均匀插值,得到多边形的光滑颜色分布。计算机图形学中的两个主要算法是 Gouraud 明暗处理和 Phong 明暗处理,它们分别采用双线性光强插值算法和双线性法向插值算法。

1) 双线性光强插值(Gouraud 明暗处理)

Gouraud 明暗处理的基本思想是:先计算多边形各个顶点的光强,然后通过双线性插

值，计算出多边形内各点的光强。其基本的算法步骤为：

(1) 计算多边形顶点的平均法向量；

(2) 根据基本光照模型计算顶点的平均光强；

(3) 通过线性插值，计算多边形边上的各点光强；

(4) 通过线性插值，计算多边形内各点的光强。

假定我们用多面体来近似表示某一个曲面体，已知多面体的各个多边形表面的顶点和法向量，假设顶点 A 相邻的多边形有 k 个，其法向量分别为 $\boldsymbol{N}_i$，$i=0,1,\cdots,k-1$，则顶点 A 的单位法向量为

$$\boldsymbol{N}_a = \frac{\sum_{i=0}^{k-1}\boldsymbol{N}_i}{\left|\sum_{i=0}^{k-1}\boldsymbol{N}_i\right|}$$

如图 7.2-7 所示，一扫描线与多边形的投影边界线相交于 a 和 b 两点，s 是投影于该扫描线上 a 到 b 之间的多边形采样像素点，四个顶点 V_1、V_2、V_3 和 V_4 的光亮度分别为 I_1、I_2、I_3 和 I_4；取 a 点的光亮度 I_a 为 I_1 和 I_2 的线性插值，b 点的光亮度 I_b 为 I_1 和 I_4 的线性插值，则 s 点发出的光亮度 I_s 为 I_a 和 I_b 的线性插值，即

$$I_a = uI_1 + (1-u)I_2,\ u = (y_a - y_2)/(y_1 - y_2)$$
$$I_b = vI_1 + (1-v)I_4,\ v = (y_b - y_4)/(y_1 - y_4)$$
$$I_s = tI_a + (1-t)I_b,\ t = (x_s - x_b)/(x_a - x_b)$$

双线性光强插值方法的优点是计算量小，可以得到连续的曲面光强。它的缺点是，它使得物体的高光部位变得模糊，面上的高光有时甚至出现异常形状，线性光强插值有时会造成表面出现过亮或过暗的条纹，即马赫带效应，使图形的真实感降低。由于 Gouraud 明暗绘制方法的计算量较小，并且产生的图形一般都比较好，所以，图形接口系统 OpenGL 实现的就是双线性光强插值算法。

2) 双线性法向插值(Phong 明暗处理)

Phong 明暗处理的基本思想是：多边形内各点的法向量通过对顶点的法向量作双线性插值得到，在多边形内构造一个连续变化的法向量函数，并将它代入光亮度计算公式，即得到由多边形近似表示的曲面在各采样点处的光亮度值。算法一般步骤如下(如图 7.2-8 所示)：

(1) 计算每个多边形顶点的平均单位法向量；

(2) 对多边形顶点的法向量进行双线性插值，得到多边形内每个点的法向量；

(3) 根据光照模型沿每条扫描线计算多边形内各点对应的投影像素的光强度值。

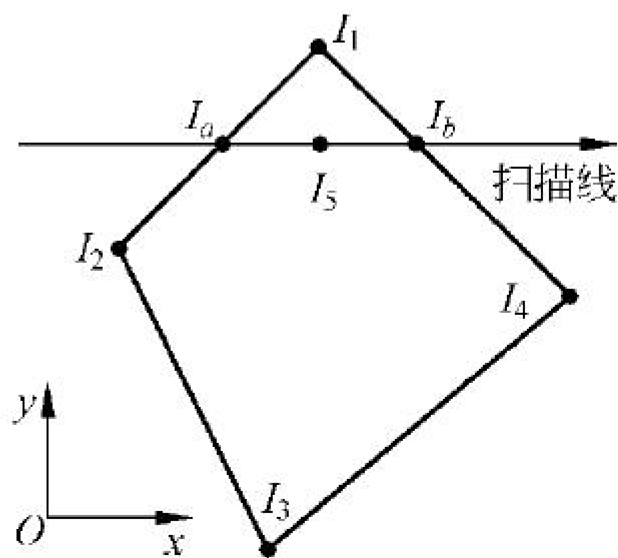


图 7.2-7 双线性光强插值

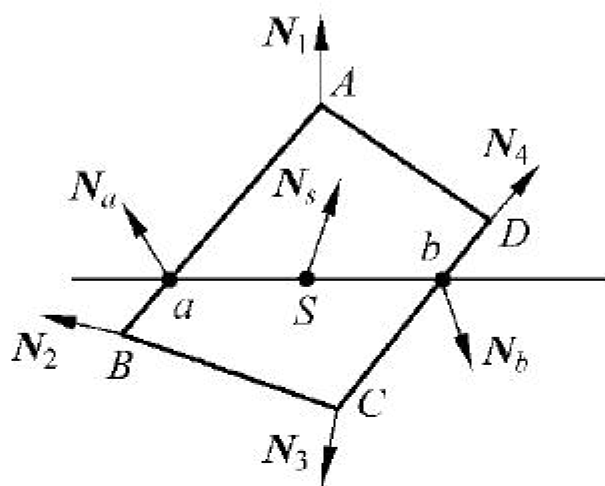


图 7.2-8 双线性法向插值

多边形内任意一点的法向量的双线性插值的计算公式与 Gouraud 明暗绘制方法中的类似,把 $\boldsymbol{I}$ 换为 $\boldsymbol{N}$,则 s 点处的法向量 $\boldsymbol{N}_s$ 就有如下的插值公式:

$$\boldsymbol{N}_a = u\boldsymbol{N}_1 + (1-u)\boldsymbol{N}_2, \quad u = (y_a - y_2)/(y_1 - y_2)$$
$$\boldsymbol{N}_b = v\boldsymbol{N}_1 + (1-v)\boldsymbol{N}_4, \quad v = (y_b - y_4)/(y_1 - y_4)$$
$$\boldsymbol{N}_s = t\boldsymbol{N}_a + (1-t)\boldsymbol{N}_b, \quad t = (x_s - x_b)/(x_a - x_b)$$

Phong 明暗处理的优点是生成的图形有明显的高光,比 Gouraud 明暗绘制方法的真实感效果更好,能缓解马赫带效应。缺点是由于每一个像素点的光亮度值都需要进行光照模型计算,故计算量较大。

8. 透明处理

自然界有很多物体是透明或者半透明的,如玻璃瓶。当光线透射穿过透明物体到达人的眼睛时,就可以清楚地看到后面的物体;同理,当光线投射穿过半透明的物体(如磨砂玻璃)到达人的眼睛时,看到的是后面模糊的景物。后面物体的反射光通过透明物体的透射到达我们的眼睛的折射光,和透明物体本身到达我们眼睛的反射光叠加,形成了最终所看到的颜色,如图 7.2-9 所示。

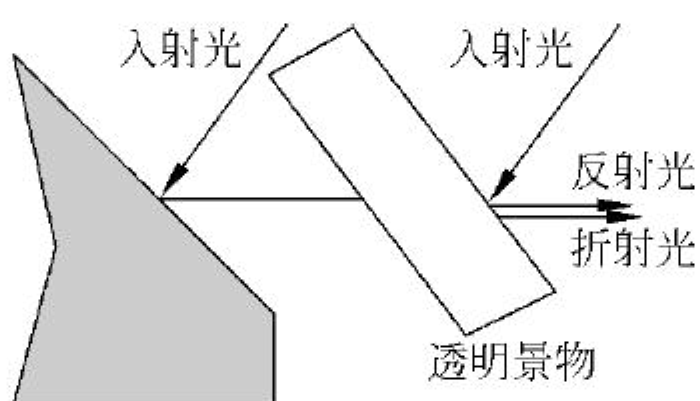


图 7.2-9 透明处理

在简单透明处理中,忽略光线在穿过透明物体时所发生的折射角度变化,在投影面上点的光强由不透明的背景物体穿过透明物体的透射光强与透明表面的反射光强加权得到:

$$I_p = (1-k_t)I_A + k_t I_B$$

式中,I_p 为投影面上点的光强;I_A 为透明表面的光强;I_B 为背景物体表面的光强;k_t 为透明物体的投射系数,$0 \leqslant k_t \leqslant 1$。

7.2.2 整体光照模型

考虑物体之间的相互影响、光在物体之间的多重吸收、反射、透射、阴影和表面纹理细节等所推出的光照模型为整体光照模型(global illumination model)。

在整体光照模型中,物体表面的入射光分为三部分:一部分来源于各个光源的直接照射;一部分包括从光源到达表面的透射光和从环境到达表面的透射光;一部分是来自其他物体的光强(如图 7.2-10 所示)。

所以,从视点观察到的物体 A 表面的光强来源于三方面的贡献:一方面是光源直接照射到 A 的表面被反射到人眼中的光产生的;另一方面是来自光源或其他物体的光经过 A 物体折射到人眼中的光产生的;还有一方面是物体 B 的表面将光反射到物体 A 的表面,再经过物体 A 的表面反射到人眼中产生的。

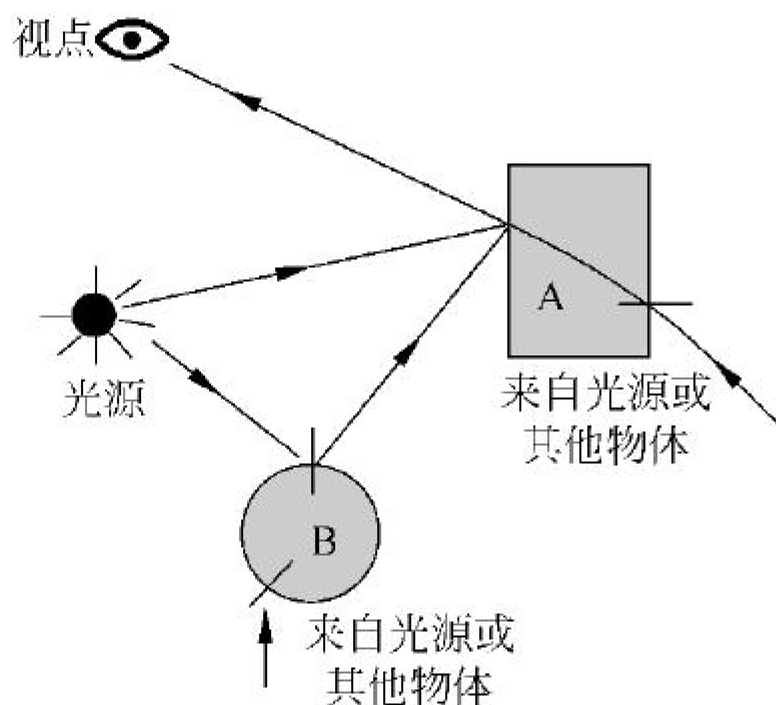


图 7.2-10 整体光照模型入射光

整体光照模型比局部光照模型复杂得多,较高级的光照模型使物体的光照效果能得以更好地表现,与实际情况非常吻合,但它需要的计算量庞大,生成时间甚长,制造成本高。

典型的整体光照明模型有 Whitted 光照模型和 Hall 光照模型。与整体光照明模型相应的算法主要有光线跟踪算法和辐射度算法，本节主要介绍 Whitted 光照模型和光线跟踪算法。

1. Whitted 光照模型

Whitted 整体光照模型是在简单光照模型中增加了环境镜面反射光和环境规则透射光两个因素。Whitted 模型基于下列假设：

从物体表面到达视点 V 的光强 I 由三部分组成：

（1）由光源直接照射引起的反射光强 I_{local}；

（2）沿视点 V 的镜面反射方向来的环境光 I_s 投射在光滑表面上产生的镜面反射光；

（3）沿视点 V 的规则透射方向来的环境光 I_t 通过透射在透明体表面上产生的规则透射光。

因此，Whitted 模型可用以下公式表述：

$$I = I_{local} + K_s I_s + K_t I_t$$

式中，K_s 为反射系数，K_t 为透射系数，$0 \leqslant K_s, K_t \leqslant 1$。

Whitted 模型中，I_{local} 可直接采用不考虑环境光的 Phone 模型计算。在计算镜面反射光及透射光时，需要计算反射方向 $\boldsymbol{r}$ 和透射方向 $\boldsymbol{t}$，如图 7.2-11 所示。

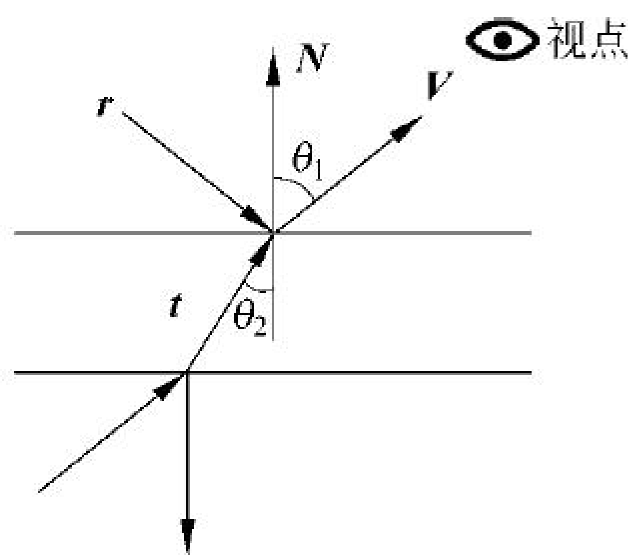


图 7.2-11　反射及投射方向

反射光的矢量：

$$\boldsymbol{r} = \boldsymbol{V} - 2(\boldsymbol{N} \cdot \boldsymbol{V})\boldsymbol{N}$$

透射光的矢量：

$$\boldsymbol{t} = \eta \boldsymbol{V} + \left(\sqrt{1 - \eta^2(1 - (\boldsymbol{V} \cdot \boldsymbol{N})^2)} - \eta(\boldsymbol{V} \cdot \boldsymbol{N})\boldsymbol{N}\right)$$

其中，η 是介质 1 的折射率 η_1 与介质 2 的折射率 η_2 之比，$\eta = \dfrac{\sin\theta_2}{\sin\theta_1} = \dfrac{\eta_1}{\eta_2}$。

2. 光线跟踪算法

光线跟踪是自然界光照明物理过程的近似逆过程，即逆向跟踪从光源发出的光经环境景物间的多次反射、折射后投射到景物表面，最终进入人眼的过程。由于这一过程只跟踪景物的镜面反射光线和规则透射光线，故是一近似过程。

算法的基本思想如下。

如图 7.2-12 所示，对于屏幕上的每个像素，跟踪一条从视点出发经过该像素的光线，求出与环境中物体的交点。在交点处光线分为两支，分别沿镜面反射方向和透明体的折射方

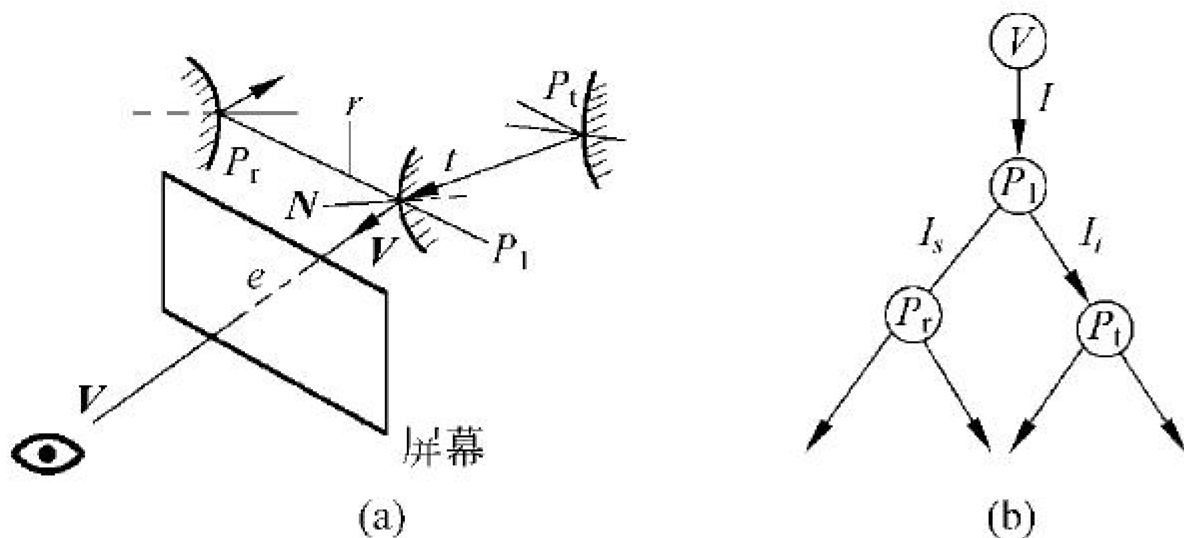


图 7.2-12　光线跟踪算法

向进行跟踪，形成一个递归的跟踪过程。光线每经过一次反射或折射，由物体材质决定的反射、折射系数都会使其强度衰减，当该光线对原像素光亮度的贡献小于给定的阈值时，跟踪过程即停止。光线跟踪的阴影处理也很简单，只需从光线与物体的交点处向光源发出一条测试光线，就可以确定是否有其他物体遮挡了该光源(对于透明的遮挡物体需进一步处理光强的衰减)，从而模拟出软影和透明体阴影的效果。

光线跟踪很自然地解决了环境中所有物体之间的消隐、阴影、镜面反射和折射等问题，能够生成十分逼真的图形，而且算法的实现也相对简单。光线与物体的求交是光线跟踪算法的核心，求交运算的效率对于整个算法的效率影响很大。

1) 光线与球求交

球是光线跟踪算法中最常用的体素，很容易进行光线与球的相交判断，球又常常用来作为复杂物体的包围盒。设(x_0, y_0, z_0)为光线的起点坐标，(x_d, y_d, z_d)为光线的方向，已经规格化。(x_c, y_c, z_c)为球心坐标，R 为球的半径。

由起点发出的光线参数方程为

$$\begin{cases} x = x_0 + x_d t \\ y = y_0 + y_d t \\ z = z_0 + z_d t \end{cases}$$

球面的隐式方程为

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = R^2$$

将光线的参数方程代入球面方程，并合并整理得

$$At^2 + Bt + C = 0$$

其中

$$\begin{aligned} A &= x_d^2 + y_d^2 + z_d^2 = 1 \\ B &= 2[x_d(x_0 - x_c) + y_d(y_0 - y_c) + z_d(z_0 - z_c)] \\ C &= (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 \end{aligned}$$

解方程得

$$t = \frac{-B \pm \sqrt{B^2 - 4C}}{2}$$

$B^2 - 4C < 0$，光线与球无交；

$B^2 - 4C = 0$，光线与球相切，$t = -B/2$；

$B^2 - 4C > 0$，光线与球有两个交点，若 $t < 0$，交点无效，将 t 代入光线的方程，得交点坐标，设坐标为(x_i, y_i, z_i)，则可计算出交点处的法向量为$\left(\frac{x_i - x_c}{R}, \frac{y_i - y_c}{R}, \frac{z_i - z_c}{R}\right)$。

已知光线和物体交点的法向量，则根据 Whitted 模型中的反射光的矢量和投射光矢量方程，即得对应的反射方向和折射方向，从而进一步迭代计算。

2) 光线与多边形求交

光线与多边形求交分两步进行：第一步，计算多边形所在的平面与光线的交点，交点计算可参考 6.3.2 节隐线算法的深度检测相关内容；第二步，判断所得的交点是否在多边形内部，可利用交点对应的扫描线与多边形投影的交点对区间来判断。

3）光线与二次曲面求交

二次曲面方程的一般形式表示为

$$F(x,y,z)=Ax^2+2Bxy+2Cxz+2Dx+Ey^2+2Fyz+2Gy+Hz^2+2Iz+J=0$$

用矩阵表示为

$$[x\quad y\quad z\quad 1]\begin{bmatrix}A & B & C & D\\ B & E & F & G\\ C & F & G & H\\ D & G & I & J\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}=0$$

将光线的参数方程代入二次曲面的一般形式得

$$at^2+bt+c=0$$

其中

$$\begin{aligned}a&=Ax_d^2+2Bx_dy_d+2Cx_dz_d+Ey_d^2+2Fy_dz_d+Hz_d^2\\ b&=2[Ax_0x_d+B(x_0y_d+x_dy_0)+C(x_0z_d+x_dz_0)+Dx_d+\\ &\quad Ey_0y_d+F(y_0z_d+y_dz_0)+Gy_d+Hz_0z_d+Iz_d]\\ c&=Ax_0^2+2Bx_0y_0+2Cx_0z_0+2Dx_0+Ey_0^2+2Fy_0z_0+\\ &\quad 2Gy_0+Hz_0^2+2Iz_0+J\end{aligned}$$

方程的解为 $t=\frac{-b\pm\sqrt{b^2-4ac}}{2}$，当 t 为实数时，将 t 代入光线参数方程可得交点坐标：

$$(x_i,y_i,z_i)=(x_0+x_d\cdot t,y_0+y_d\cdot t,z_0+z_d\cdot t)$$

从而，可计算交点处的法向量：

$$(x_n,y_n,z_n)=\left(\frac{\partial F}{\partial x},\frac{\partial F}{\partial y},\frac{\partial F}{\partial z}\right)$$

$$x_n=2(Ax_i+By_i+Cz_i+D)$$

$$y_n=2(Ax_i+Ey_i+Fz_i+G)$$

$$z_n=2(Cx_i+Fy_i+Hz_i+I)$$

大量的光线与景物的求交测试和交点计算，是导致计算开销大的主要原因。尽量减小求交计算量是提高光线跟踪效率的关键，常用的方法有包围盒、层次结构(hierarchies)及区域分割(spatial partitioning)等，本章不再赘述。

光线跟踪是一个典型的采样过程，各个屏幕像素的亮度都是分别计算的，因而会产生走样，而算法本身的计算量使得传统的加大采样频率的反走样技术难以实用。

像素细分是一种适用于光线跟踪的反走样技术，具体方法是：首先对每一像素的角点用光线跟踪计算亮度；然后比较各角点的亮度，若差异较大，则将像素细分为4个子区域，并对新增的5个角点用光线跟踪计算亮度；重复比较与细分，直到子区域各角点亮度差异小于给定的阈值为止；最后加权平均求出像素点的显示亮度。

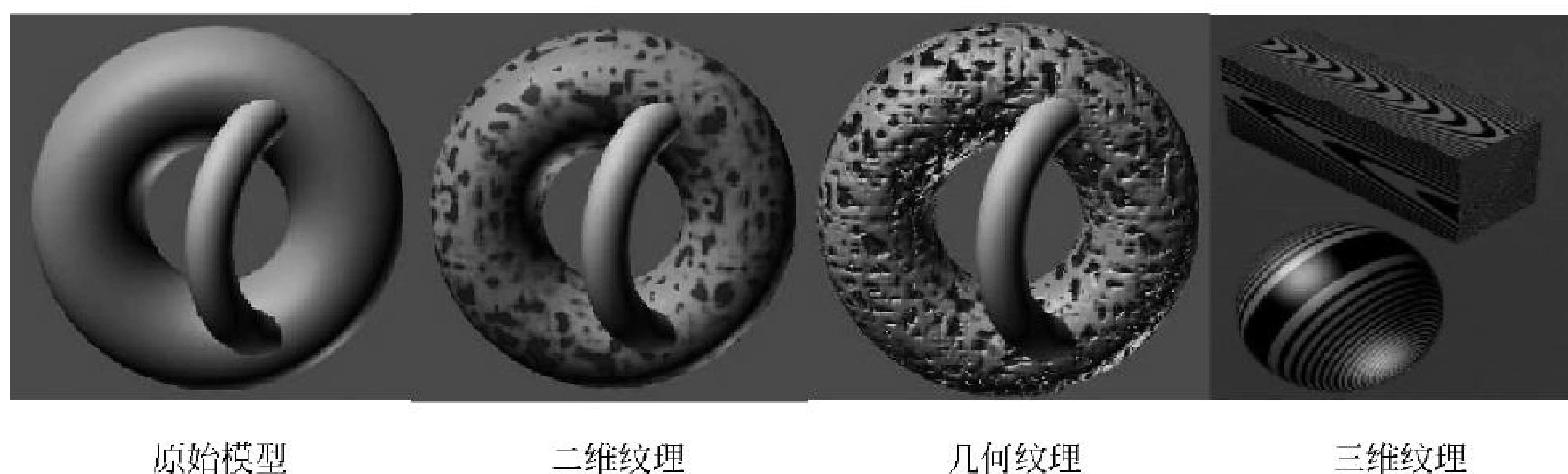
光线跟踪的另一个问题是，光线都是从视点发出的，阴影测试光线则需另外处理，因而无法处理间接的反射或折射光源，例如镜子或透镜对光源所产生的作用就难以模拟。为解决这一问题，可以从光源和视点出发对光线进行双向跟踪。但是，大量从光源出发的光线根本不可能到达屏幕，这使得双向光线跟踪的计算量显著增大，难以实用。

7.3 纹　　理

7.3.1 概述

我们观察周围的景物，会发现现实世界中的大部分物体，其表面往往有各种表面细节和图案花纹，这就是通常所说的纹理，例如，建筑物墙壁上的装饰图案、木材表面的木纹以及橘子皮表面的皱纹等。前面介绍的光照明模型生成的真实感图像，由于表面过于光滑单调，反而显得不真实，所以更精确的真实感图形绘制还要考虑物体表面的细节纹理。

纹理是物体表面的细小结构，有颜色纹理和几何纹理两种类型。颜色纹理通过颜色色彩或明暗度的变化体现表面细节，它一般是二维图像纹理，如物体表面花纹、图案等，也有三维纹理，如木材纹理等，颜色纹理取决于物体表面的光学属性。几何纹理主要是由于不规则的细小凹凸造成的表面细小结构，它由物体表面的微观几何形状决定。各种类型的纹理如图 7.3-1 所示。



原始模型　二维纹理　几何纹理　三维纹理

图 7.3-1　不同类型的纹理

纹理映射是把纹理图像值映射到三维物体表面的技术，并在应用光照模型时将这些花纹的颜色考虑进去。如图 7.3-2 所示，将一个平面花纹图案按照一定的函数关系绘制在一个水壶旋转的表面上，即为纹理映射。这样，对物体表面细节的模拟使绘制的图形更接近自然景物。



图 7.3-2　纹理映射

有两种纹理定义的方法：

(1) 图像纹理　将二维纹理图案映射到三维物体表面，绘制物体表面上一点时，采用相

应的纹理图案中相应点的颜色值。

(2) 函数纹理　用数学函数定义简单的二维纹理图案，如方格地毯。或用数学函数定义随机高度场，生成表面粗糙纹理即几何纹理。

在进行纹理和物体表面之间的映射时，通过建立它们之间的某种一一对应关系来实现。例如，纹理空间的参数和物体表面的参数之间建立一一对应关系，那么对应参数处的纹理就映射到物体表面上。在光照模型中，也可以通过改变漫反射系数来改变物体的颜色或者改变物体表面的法向量来产生纹理效果。

7.3.2　二维纹理映射和三维纹理映射

二维纹理映射实质上是从二维纹理平面到三维景物表面的一个映射。通常，二维纹理在一个纹理空间(s,t)坐标系内的矩形区域中用光强度值来定义，其中每一点处，均定义有一灰度值或颜色值；而场景中的物体表面是在景物空间(u,v)坐标系中定义的，投影平面上的像素点是在图像空间内的直角坐标系 xOy 中定义的。在绘制时，应用纹理映射方法可以方便地确定景物表面上任一可见点 P 处在纹理空间中的对应位置(s,t)，而(s,t)处所定义的纹理值或颜色值即描述了景物表面在 P 处的某种纹理属性；把该点处的纹理颜色值作为漫反射系数代入光照模型中进行计算，最后，就能够生成具有纹理效果的真实感图像。

二维纹理最常见的表示方法就是数字化的彩色图像，其分辨率用 $m\times n$ 表示，颜色数用 $2k$ 表示，例如，$k=1$ 表示黑白(二值)图像；用位图(bitmap)文件(BMP 文件)或其他格式的图像文件(例如：PCX、TIFF 或 JPEG 等图像文件)保存。

纹理的另外一种表示方法就是用解析函数或过程表示。实际上，许多纹理都可以用二维纹理函数 $t(u,v)$或函数过程来表示。

实现纹理映射的方法主要是从纹理空间→景物空间→图像空间，如图 7.3-3 所示。为了简化计算，由纹理空间向景物空间的映射可以采用线性映射：

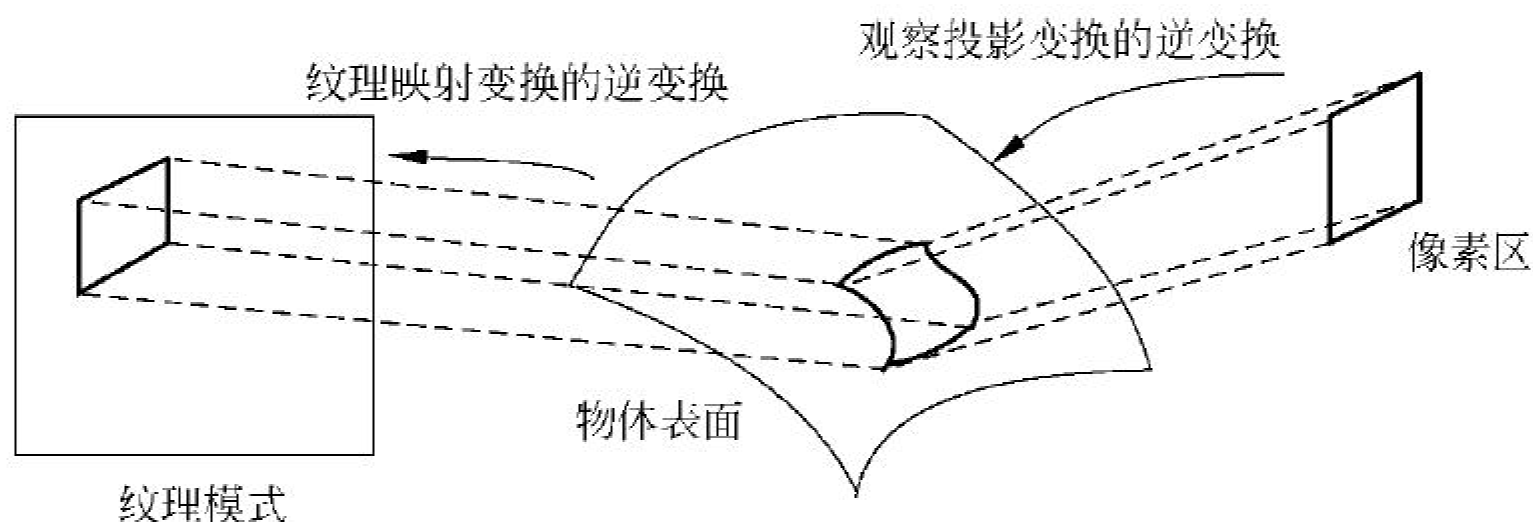


图 7.3-3　二维纹理映射

在纹理空间，纹理图案在坐标为(s,t)的二维空间中描述。在物体空间用参数(u,v)表示物体的表面。纹理空间到物体空间的映射函数为 $u=f_u(s,t)$，$v=f_v(s,t)$。通常，假设映射函数是线性的，使得纹理图案可以按比例伸缩，所以，函数可以写成 $u=As+B$，$v=Ct+D$。

例 7.1　给定一个线性方程，以此把规范化网格图案映射到第一象限球面的下边部分，如图 7.3-4 所示。

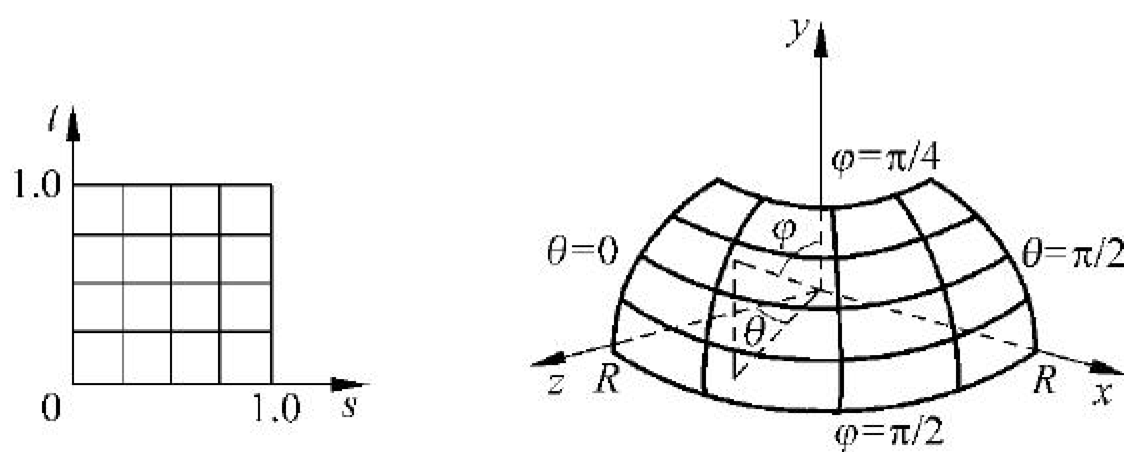


图 7.3-4 网格图案映射

解 设球表面的参数为

$$(u,v)=(\theta,\varphi),\quad 0\leqslant\theta\leqslant\pi/2,\pi/4\leqslant\varphi\leqslant\pi/2$$

在三维直角坐标系中,球表面的参数方程为

$$\begin{cases}x=r\cos u\sin v\\y=r\sin u\sin v\\z=r\cos v\end{cases}$$

映射图案和该球面的映射关系为

$$s=0,\quad t=0\Rightarrow u=0,\quad v=\pi/2$$
$$s=1,\quad t=0\Rightarrow u=\pi/2,\quad v=\pi/2$$
$$s=0,\quad t=1\Rightarrow u=0,\quad v=\pi/4$$
$$s=1,\quad t=1\Rightarrow u=\pi/2,\quad v=\pi/4$$

按照映射关系 $u=As+B,v=Ct+D$ 计算系数,得

$$\begin{cases}u=\dfrac{\pi}{2}s\\v=\dfrac{\pi(2-t)}{4}\end{cases}$$

当采用某种真实感图形绘制算法(如:光线跟踪)生成该球面的真实感图形时,将每个像素区域所覆盖的矩形纹理图案中的光亮度值取平均,把该平均值作为球表面上 P 点的漫反射系数 K_d 代入光照明模型进行计算,最后就能够生成带有纹理图案的球面的真实感图形。

投影纹理和映射纹理的缺点主要是二维纹理和三维物体表面之间的不匹配,这样会造成纹理变形,而且不能保证纹理的连续性。

假如在三维物体空间中,物体中每一个点(x,y,z)均有一个纹理值 $t(x,y,z)$,其值由纹理函数 $t(x,y,z)$唯一确定,那么对于物体上的空间点,就可以映射到一个纹理空间上了,而且是三维的纹理函数,这是提出三维纹理的基本思想。三维纹理映射的纹理空间定义在三维空间上,与物体空间是同维的,在纹理映射时,只需把场景中的物体变换到纹理空间的局部坐标系中即可。

第 8 章

曲线曲面

真实世界物体的形状除了我们常见的直线、圆、椭圆以及平面、圆柱面、球面、圆锥面和圆环面这些基本线面图形(又称初等解析曲线曲面)外,还有很多复杂的形状,如飞机、轮船、潜艇以及汽车的外形等,这些形状不能用基本图形表示,我们称之为自由曲线曲面(又称高等曲线曲面)。自由曲线曲面也是计算机图形学的一个重要研究内容,从计算机图形学的应用全局看,自由曲线曲面造型具有重要的作用,这是因为传统意义下的造型技术至今还限制在操作圆锥体、椭球体等规则曲面形体,而地形地貌描述、矿藏储量图示、铁路勘察设计与环境工程、人体器官造型与CT图像三维重建、服装设计、制鞋、虚拟视景生成等都要用到不规则曲面的拟合和生成技术。这些问题的覆盖域要宽广得多,求解的技术难度也更大。由于自由曲线曲面的研究理论和方法有一定的系统化和独立性,因此,又形成了一门新的学科——计算机辅助几何设计(computer aided geometric design,CAGD),曲线曲面研究的结果使真实世界的形状具有统一的数学模型——非均匀有理B样条曲线曲面(nonuniform rational B-spline,NURBS)。关于曲线曲面详细的理论请参考相关的书籍,本章主要讲述曲线曲面的基础理论、常用曲线曲面的表达方法以及相关实践等。

8.1 曲线曲面基础知识

8.1.1 曲线和曲面的表示方法

曲线曲面可以用显式方程、隐式方程和参数方程等形式表示。

显式方程表示形式如

$$y = f(x)$$

例如,直线方程 $y=kx+b$,二次曲线方程 $y=ax^2+bx+c$ 等。

隐式方程表示形式如

$$f(x,y) = 0$$

例如,直线表示为 $ax+by+c=0$,圆表示为 $x^2+y^2=R^2$ 等。

显式或隐式表示方法可以非常直观地表示初等解析曲线曲面,曲线曲面的特性通过表示形式能够清楚地表现出来。但是,对于高等曲线曲面,显式和隐式表示方法具有很大的局限性。首先,对于高等曲线曲面,坐标之间的关系不能用简单的显式或者隐式方程来表示,即使能够表示,由于它们与坐标系严格相关,当坐标系统发生了变化,那么函数关系也随之

变化；另外，这种表示方法也不便于计算机编程，会出现斜率为无穷大的情形（如垂线），造成程序不稳定。

在几何造型系统中，曲线曲面常用参数形式表示，曲线曲面上任一点的每一个坐标分量均表示成给定参数的函数。假定用 t 表示参数，平面曲线上任一点 P 可表示为

$$P(t)=[x(t),y(t)]$$

空间曲线上任一三维点 P 可表示为

$$P(t)=[x(t),y(t),z(t)]$$

例如，直线段参数方程

$$P(t)=P_1+(P_2-P_1)t$$

其中，P_1、P_2 为直线两个端点，$t\in[0,1]$。

参数表示具有如下优点：

(1) 可以满足几何不变性的要求，和坐标系无关；

(2) 有更大的自由度来控制曲线、曲面的形状；

(3) 对曲线、曲面进行变换，可对其参数方程直接进行几何变换；

(4) 便于处理斜率为无穷大的情形，不会因此而中断计算；

(5) 便于用户把低维空间中的曲线、曲面扩展到高维空间去；

(6) 规格化的参数变量 $t\in[0,1]$，使其相应的几何分量是有界的，而不必用另外的参数去定义边界；

(7) 易于用矢量和矩阵表示几何分量，简化计算。

8.1.2 连续性、样条及曲线曲面构造方式

假定参数曲线段 P_i 以参数形式进行描述：

$$p_i=p_i(t),\quad t\in[t_{i0},t_{i1}]$$

0 阶参数连续性——若两个相邻的曲线段在首末点相连接，即 $p_i(t_{i1})=p_{(i+1)}(t_{(i+1)0})$，则称两个曲线段 C^0 连续，记为 0 阶参数连续，如图 8.1-1(a)所示。

1 阶参数连续性——若两个相邻曲线段在相交点处有相同的一阶导数，即 $p_i(t_{i1})=p_{(i+1)}(t_{(i+1)0})$，$p'_i(t_{i1})=p'_{(i+1)}(t_{(i+1)0})$，则称两曲线段 C^1 连续，记为 1 阶参数连续，如图 8.1-1(b)所示。

2 阶参数连续性——若两个相邻曲线段的方程在相交点处具有相同的一阶和二阶导数，则称两曲线段 C^2 连续，记为 2 阶参数连续，如图 8.1-1(c)所示。

从图 8.1-1 可以看出，当组合的相邻曲线达到 2 阶连续时，曲线段之间已经具有了非常好的光滑连接。

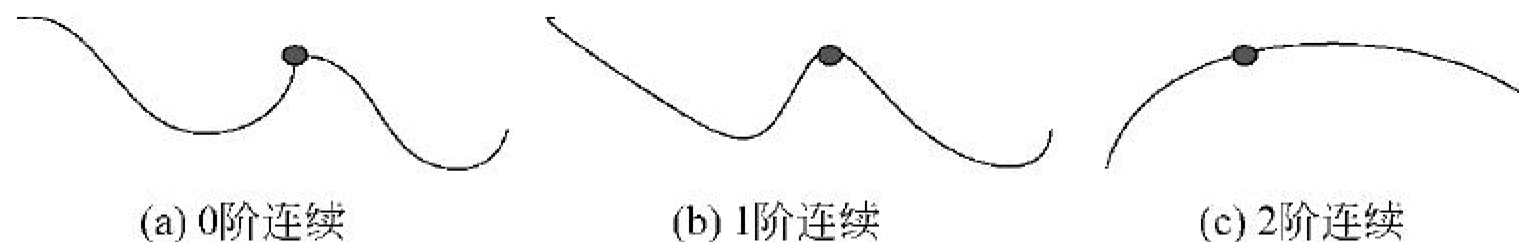


图 8.1-1 曲线连续性

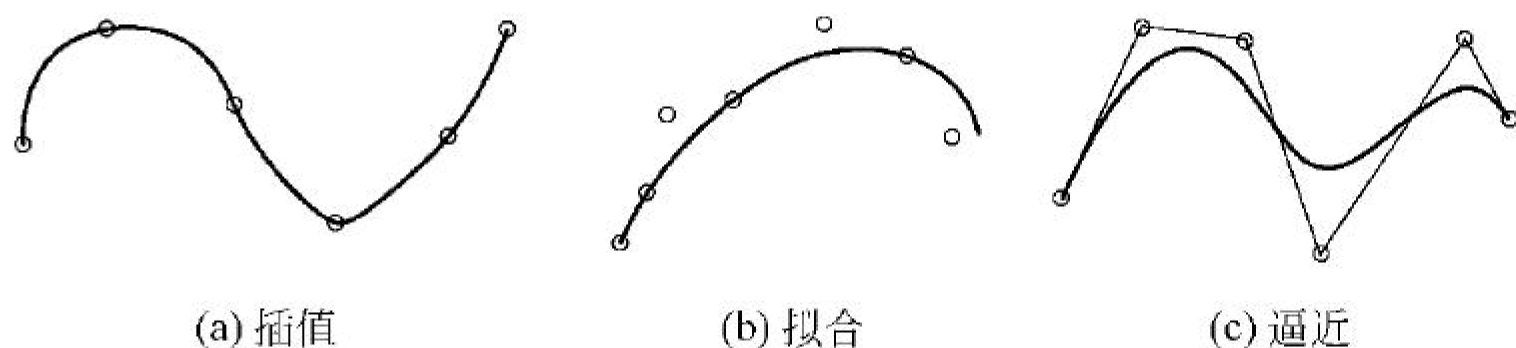
样条概念源于生产实践。“样条”是绘制曲线的一种绘图工具，是富有弹性的细长条。绘图时用压铁使样条通过指定的形值点（样点），并调整样条使它具有满意的形状，然后沿样条画出曲线。样条曲线是指由多项式曲线段连接而成的曲线，在每段的边界处满足特定的连续条件。样条曲面则可以用两组正交样条曲线来描述。

用数学方法构造自由曲线和曲面有三种方式。

（1）插值：用光滑的曲线或曲面把给定的若干个离散点连接起来，即由型值点构造的曲线或曲面通过所有的型值点，如图 8.1-2(a)所示。

（2）拟合：仅要求比较贴近给定的型值点来构造曲线，并不要求曲线通过全部给定的型值点，如图 8.1-2(b)所示。

（3）逼近：先给定由若干个型值点勾画的一条折线轮廓，然后要求用曲线、曲面逼近这个折线轮廓，这一折线轮廓构成的多边形称为控制多边形，如图 8.1-2(c)所示。



(a) 插值　　(b) 拟合　　(c) 逼近

图 8.1-2　三种曲线曲面构造方式

插值曲线具有计算简单可靠等优点，常作为飞机、船舶、汽车外形的数学放样工具。但在外形的设计中，初始给出的特征点往往不是精确的数据点，当然也不要求对这些本来很粗糙的特征点进行精确的插值，而且作为外形设计工具来说，样条插值曲线还缺乏灵活性和直观性，不够方便。

通常的设计方法往往是一个交互式的过程，即设计人员根据基本的设计要求及原始型值点进行初步设计，然后逐次把所得到的结果和设计要求进行对比，找出不足之处，加以改进，最终达到满意的结果。

目前，在曲线、曲面形状设计中，较为成功的方法是所谓控制点造型法。其中最重要而又有代表性的曲线曲面生成方法是 Bézier 曲线曲面、B 样条曲线曲面、NURBS 表示法等。

8.2　Bézier 曲线曲面

8.2.1　Bézier 曲线定义

法国雷诺汽车公司的工程师 Bézier 在 20 世纪 60 年代开始研究在“逼近”意义上的曲线曲面的参数表示法，这种自由曲线曲面的设计方法被称为 Bézier 方法。Bézier 方法是一种只用一组控制点就可以确定曲线曲面形状的方法，该方法直观性很强，并将函数逼近同几何表示结合起来，使得设计师在计算机上就像使用作图工具一样得心应手。Bézier 曲线有如下特点：一是只需给出数据点就可以构造曲线，不要求给出导数；二是 Bézier 曲线的阶次严格依赖于确定该段曲线的数据点个数，因而不同的段可以是不同次的曲线；三是 Bézier

曲线虽不完全通过给定的数据点，但这些点控制着曲线的形状，曲线与数据点构成的折线间有着直观的形状对应关系。

通过 $n+1$ 个顶点定义的 Bézier 曲线的 n 次多项式参数方程为

$$P(t)=\sum_{i=0}^{n}p_iB_{i,n}(t),\quad 0\leqslant t\leqslant 1$$

式中，p_i 为各顶点的位置矢量；$B_{i,n}(t)$为伯恩斯坦(Bernstein)基函数，它决定了在不同 t 值下各顶点位置矢量对整段曲线所起作用的大小。$B_{i,n}(t)$表示为

$$B_{i,n}(t)=\frac{n!}{i!(n-i)!}t^i(1-t)^{n-i},\quad i=0,1,2,\cdots,n$$

注意：当 $i=0$ 时，$t=0,0^0=1,0!=1$。

顺序连接各顶点构成一个开口多边形，此多边形也称为控制多边形或者特征多边形。图 8.2-1 所示为三次 Bézier 曲线以及对应的控制多边形。

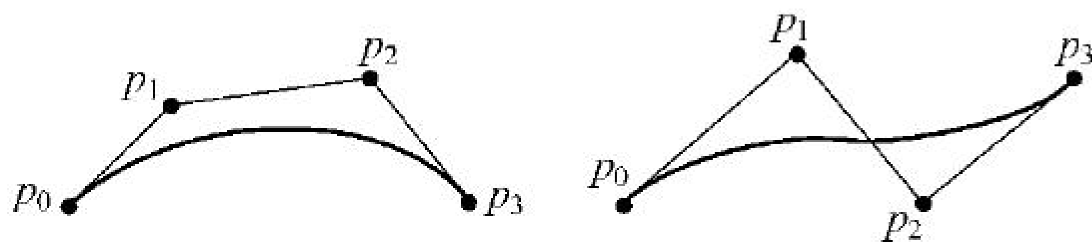


图 8.2-1　三次 Bézier 曲线以及控制多边形

8.2.2 Bézier 曲线的性质

性质一，Bézier 曲线的端点性质。由 Bézier 曲线的定义可得

$$P(0)=\sum_{i=0}^{n}p_iB_{i,n}(0)=\frac{n!}{0!(n-0)!}\cdot 0^0\cdot(1-0)^n p_0=p_0$$

$$P(1)=\sum_{i=0}^{n}p_iB_{i,n}(1)=\frac{n!}{n!(n-n)!}\cdot 1^n\cdot(1-1)^{n-n}p_n=p_n$$

由此可见，曲线通过控制多边形的起点和终点。对 $P(t)$求导，因为

$$\begin{aligned}B'_{i,n}(t)&=\frac{n!}{i!(n-i)!}[it^{i-1}(1-t)^{n-i}-(n-i)t^i(1-t)^{n-i-1}]\\&=n\left[\frac{(n-1)!}{(i-1)!(n-i)!}t^{i-1}(1-t)^{n-i}-\frac{(n-1)!}{i!(n-i-1)!}t^i(1-t)^{n-i-1}\right]\\&=n[B_{i-1,n-1}(t)-B_{i,n-1}(t)]\end{aligned}$$

则

$$P'(t)=n\sum_{i=0}^{n}[B_{i-1,n-1}(t)-B_{i,n-1}(t)]p_i$$

整理可得

$$t=0,P'(0)=n(p_1-p_0);\quad t=1,P'(1)=n(p_n-p_{n-1})$$

这说明，曲线在起点处与控制多边形的第一边相切，在终点处与控制多边形的最后一边相切。

性质二，对称性。若保持某 Bézier 曲线特征多边形各顶点位置不变，而将它们的顺序全部颠倒过来，则所得的新曲线与原曲线相同，但走向相反。此性质由基函数可以导出：

$$B_{i,n}(t) = \frac{n!}{i!(n-i)!}t^i(1-t)^{n-i} = B_{n-i,n}(1-t)$$

这说明如果型值点的位置不变，但完全颠倒它们的顺序，即 $p_i \leftrightarrow p_{n-i}$，此时 $t \leftrightarrow 1-t$，曲线仍不变，只不过曲线的走向相反而已，如图 8.2-2 所示。

性质三，几何不变性与多值性。几何特性不随一定的坐标变换而变换的性质称为几何不变性。由 Bézier 曲线的定义可知，Bézier 曲线具有几何不变性。曲线形状仅由控制多边形的顶点决定，而与坐标系的选择无关。还可表示多值形状，如第一控制点与最后一个控制点重合时，曲线即是封闭的。

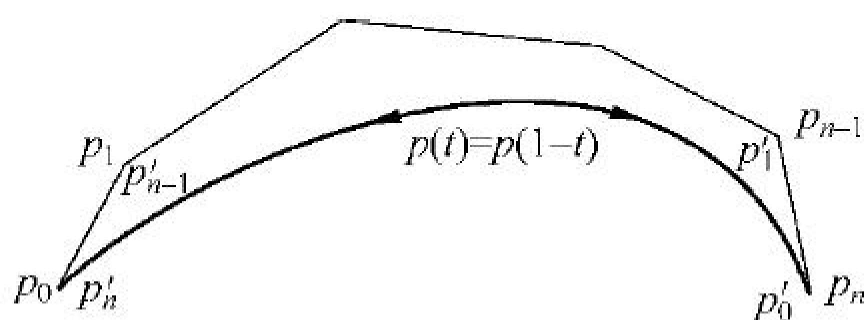


图 8.2-2　对称性

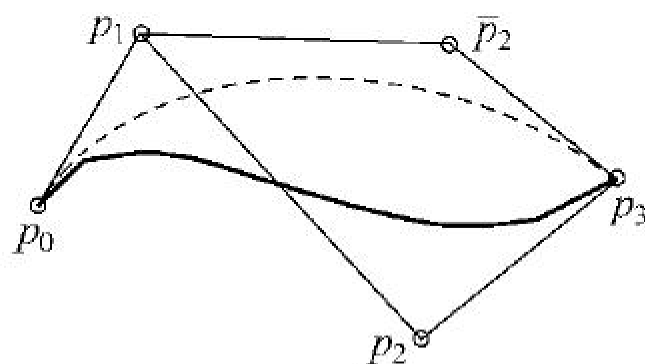


图 8.2-3　全局性控制可交互性

性质四，全局性控制（可交互性）。控制多边形各顶点大致地勾画出 Bézier 曲线的形状，要改变曲线的形状，只要改变多边形各顶点的位置即可。把控制多边形作为曲线输入和人机交互的手段，既直观又简便，即使不了解 Bézier 曲线的数学定义的人也能得心应手地使用，因此 Bézier 曲线有很好的交互性能。但是，移动任何一个控制点，都会使整段 Bézier 曲线随着变动。这也是 Bézier 曲线的一个缺点，因为它排除了对一段 Bézier 曲线作局部修改的可能。

性质五，凸包性。对区间(0,1)上的任一 t 值，点 $P(t)$ 必落在由特征多边形顶点所张成的凸包内。即当特征多边形为凸时，Bézier 曲线也为凸，Bézier 曲线的凸凹与特征多边形相一致，且在其凸包范围内，如图 8.2-4 所示。

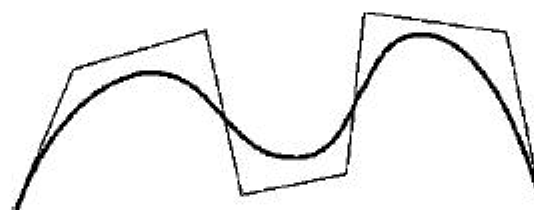


图 8.2-4　凸包性

8.2.3　低次 Bézier 曲线及矩阵表示

当 $n=1$ 时，有起点 p_0 和终点 p_1 两个控制顶点，是一次 Bézier 曲线：

$$P(t) = \sum_{i=0}^{1} p_i B_{i,1}(t) = (1-t)p_0 + tp_1$$

上式说明，一次 Bézier 曲线是连接起点 p_0 和终点 p_1 的一条直线段。

一次 Bézier 曲线用矩阵形式表示为

$$P(t) = \begin{bmatrix} t & 1 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \end{bmatrix}$$

当 $n=2$ 时，有三个控制顶点 p_0、p_1、p_2，是二次 Bézier 曲线：

$$\begin{aligned} P(t) &= \sum_{i=0}^{2} p_i B_{i,2}(t) = (1-t)^2 p_0 + 2t(1-t)p_1 + t^2 p_2 \\ &= (t^2 - 2t + 1)p_0 + (-2t^2 + 2t)p_1 + t^2 p_2 \end{aligned}$$

二次 Bézier 曲线是一条抛物线，p_0 和 p_2 是抛物线的两个端点。

二次 Bézier 曲线写成矩阵形式为

$$P(t)=\begin{bmatrix}t^2 & t & 1\end{bmatrix}\begin{bmatrix}1 & -2 & 1\\ -2 & 2 & 0\\ 1 & 0 & 0\end{bmatrix}\begin{bmatrix}p_0\\ p_1\\ p_2\end{bmatrix}$$

当 $n=3$ 时，则得三次 Bézier 曲线：

$$\begin{aligned}P(t)&=\sum_{i=0}^{3}p_iB_{i,3}(t)=p_0B_{0,3}(t)+p_1B_{1,3}(t)+p_2B_{2,3}(t)+p_{3,3}B_{3,3}(t)\\&=p_0\cdot\frac{3!}{0!3!}t^0(1-t)^3+p_1\cdot\frac{3!}{1!(3-1)!}t^1(1-t)^{3-1}+\\&\quad p_2\cdot\frac{3!}{2!(3-2)!}t^2(1-t)^{3-2}+p_3\cdot\frac{3!}{3!(3-3)!}t^3(1-t)^{3-3}\\&=(1-t)^3p_0+3t(1-t)^2p_1+3t^2(1-t)p_2+t^3p_3\\&=(-t^3+3t^2-3t+1)p_0+(3t^3-6t^2+3t)p_1+(-3t^3+3t^2)p_2+t^3p_3\end{aligned}$$

三次 Bézier 曲线用矩阵表示为

$$P(t)=\begin{bmatrix}t^3 & t^2 & t & 1\end{bmatrix}\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 3 & 0 & 0\\ 1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}p_0\\ p_1\\ p_2\\ p_3\end{bmatrix}$$

8.2.4 Bézier 曲线的拼接

Bézier 曲线方程的次数随控制点的增多而增高，如九个控制点对应八次 Bézier 曲线，这对曲线的分析、计算和稳定性都带来问题。避免高次曲线的办法是用多段低次 Bézier 曲线拼接成整条样条曲线。一般情况下 C^2 连续的三次曲线已相当理想，下面以两段三次 Bézier 曲线为例分析在连接处的连接条件。

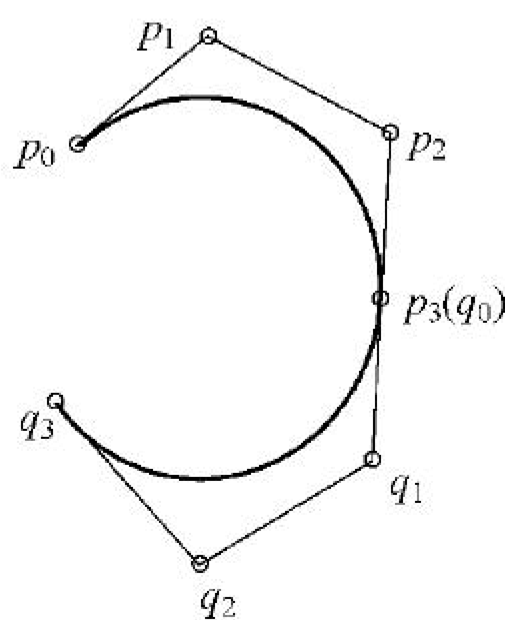


图 8.2-5 拼接一次连续

设有两个控制多边形 $p_0p_1p_2p_3$ 和 $q_0q_1q_2q_3$，要求它们所决定的两条 Bézier 曲线段在点 p_3（即 q_0 点）连续。

当两曲线段满足 C^1 即 1 阶连续时：

$$P'(1)=3(p_3-p_2),\quad Q'(0)=3(q_1-q_0)$$

令 $Q'(0)=gP'(1)$，g 是比例因子。在连接点，一阶导矢连续，即

$$q_1-q_0=g(p_3-p_2)$$

这就要求 p_2，$p_3(q_0)$，q_1 共线，且 q_1 和 p_2 在 $p_3(q_0)$ 两侧。

当两曲线段满足 C^2 连续即 2 阶连续时：

$$P''(1)=6(p_3-2p_2+p_1),\quad Q''(0)=6(q_0-2q_1+q_2)$$

$Q''(0)=P''(1)$，即

$$p_3-2p_2+p_1=q_0-2q_1+q_2$$

由于一阶连续，$q_1-q_0=g(p_3-p_2)$，则 $q_2-q_1=p_1-p_2+(1+g)(p_3-p_2)$，表明 q_2、q_1

在 p_1、p_2、p_3 所决定的平面上，故

$$q_2 - p_1 = p_3 - q_0 + 2(q_1 - p_2) = 2(q_1 - p_2)$$

即 q_2p_1 与 q_1p_2 平行，且长度是它的两倍。

8.2.5 Bézier 曲线的递推生成算法

计算 Bézier 曲线上的点除了利用 Bézier 曲线方程外，还可以使用 de Casteljau 提出的递推算法实现。

由 $n+1$ 个控制点 $p_i(i=0,1,2,\cdots,n)$ 定义的 n 次 Bézier 曲线 p_0^n，可分别由前、后 n 个控制点定义的两条 $n-1$ 次 Bézier 曲线 p_0^{n-1} 与 p_1^{n-1} 的线性组合实现：

$$p_0^n = (1-t)p_0^{n-1} + tp_1^{n-1}, \quad t \in [0,1]$$

由此得到 Bézier 曲线的递推计算公式

$$p_i^k = \begin{cases} p_i, & k = 0 \\ (1-t)p_i^{k-1} + tP_{i+1}^{k-1}, & k = 1,2,\cdots,n, i = 0,1,2,\cdots,n-k \end{cases}$$

式中，p_i 是定义 Bézier 曲线的控制点，p_0^n 为曲线 $P(t)$ 上具有参数 t 的点。这就是 de Casteljau 算法，在给定参数下，用这一递推公式求 Bézier 曲线上一点 $P(t)$ 非常有效。de Casteljau 算法稳定可靠，直观简便，可以编出十分简捷的程序，是计算 Bézier 曲线的基本算法和标准算法。

这一算法也可用简单的几何作图来实现，给定参数 $t\in[0,1]$，就把定义域分成长度为 t 和 $1-t$ 的两段。依次对原始控制多边形每一边执行同样的定比分割，所得分点就是由第一级递推生成的中间顶点 $p_i^1(i=0,1,2,\cdots,n-1)$，对这些中间顶点构成的控制多边形再执行同样的定比分割，得第二级中间顶点 $p_i^2(i=0,1,2,\cdots,n-2)$。重复进行下去，直到 n 级递推得到一个中间顶点 p_0^n，即为所求曲线上的点 $P(t)$。如图 8.2-6 所示，$t=1/3$ 时，利用同样的参数 t 值，依次作出一次 Bézier 曲线的对应点 p_0^1、p_1^1、p_2^1，再根据一次 Bézier 曲线点，作出二次 Bézier 曲线点 p_0^2、p_1^2，同理，根据二次 Bézier 曲线点，得到三次 Bézier 曲线的对应点 p_0^3。

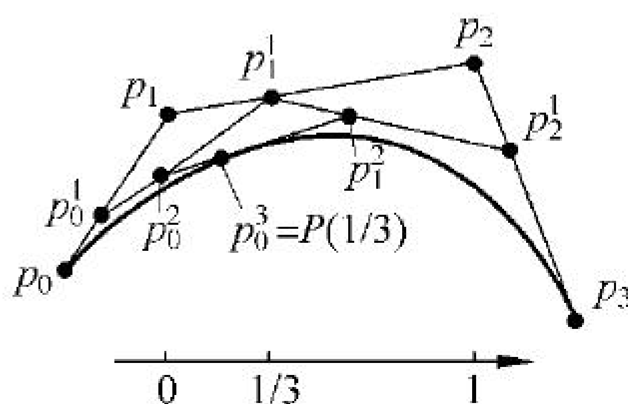


图 8.2-6 Bézier 曲线递推法几何作图

8.2.6 Bézier 曲面

设 $p_{ij}(i=0,1,2,\cdots,n,j=0,1,2,\cdots,m)$ 为 $(n+1)\times(m+1)$ 个空间点列，则 $m\times n$ 次 Bézier 曲面定义为

$$P(u,w) = \sum_{i=0}^{m}\sum_{j=0}^{n} B_{i,m}(u)B_{j,n}(w)p_{ij}, \quad 0 \leqslant u \leqslant 1, 0 \leqslant w \leqslant 1$$

其中

$$B_{i,m}(u) = C_m^i u^i (1-u)^{m-i}, \quad B_{j,n}(w) = C_n^j w^j (1-w)^{n-j}$$

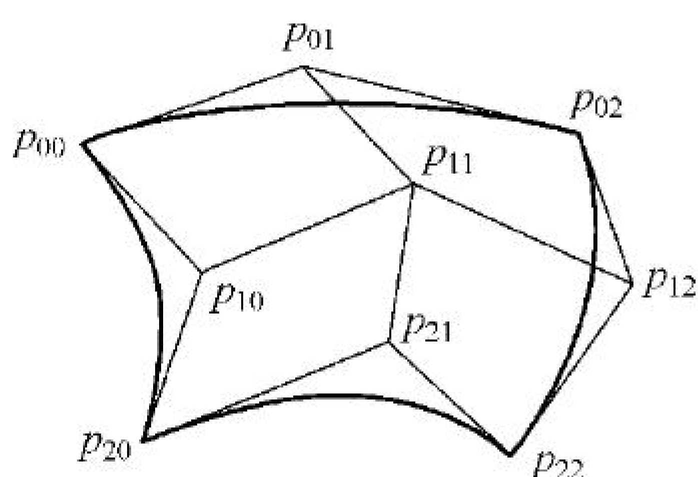


图 8.2-7 Bézier 曲面及特征网络

是伯恩斯坦基函数。依次用线段连接点列 p_{ij} 中的相邻两点形成的空间网格称为特征网格。图 8.2-7 所示为一个 2×2 次 Bézier 曲面以及对应的特征网格。

Bézier 曲面也有和 Bézier 曲线类似的端点性质、凸包性、几何不变性等特性，除此之外，还有边界线性质、端点切平面以及端点法向等性质。Bézier 曲面片拼接时，为了达到连续性也需要满足相关的条件。此处不再赘述。

8.3 B 样条曲线曲面

8.3.1 B 样条的一般定义

Bézier 曲线虽然有许多优点，但也有明显的不足之处，主要有：一是缺乏局部修改的能力，移动一个顶点的位置将会影响整条曲线的形状；二是曲线的次数受多边形点数的限制，对 Bézier 曲线曲面的拼接要达到二阶连续还需附加一些条件，比较复杂。

为克服上述缺点，专家学者们在研究 Bézier 方法的基础上构造了 B 样条曲线，在保留 Bézier 方法优点的同时，克服了 Bézier 方法的弱点。B 样条曲线是分段连续的多项式曲线，曲线的次数与控制点的个数无关，控制点只影响曲线的局部性质，B 样条曲线表示中由 B 样条基函数代替 Bézier 曲线中的 Bernstein 基函数。

给定 $m+n+1$ 个顶点 $p_i(i=0,1,\cdots,n,\cdots,m+n)$，定义 $m+1$ 段 n 次的参数曲线为

$$P_{m+1,n}(t) = \sum_{k=0}^{n} p_{m+k}F_{k,n}(t)$$

其中，$F_{k,n}(t)(k=0,1,2,\cdots,n)$称为 n 次 B 样条基函数，$0\leqslant t\leqslant 1$，其形式为

$$F_{k,n}(t) = \frac{1}{n!}\sum_{j=0}^{n-k} (-1)^j C_{n+1}^j (t+n-k-j)^n,\quad C_{n+1}^j = \frac{(n+1)!}{j!(n+1-j)!}$$

8.3.2 二次和三次 B 样条曲线段

在实际应用中，用得最多的是三次 B 样条曲线，它既能保证曲线 C^2 连续，而且计算量又不太大；其次是二次 B 样条曲线。高于三次的 B 样条曲线一般是不用的。

对于二次 B 样条曲线段(此时 $n=2$)，首先计算 B 样条基函数：

$$\begin{aligned} F_{0,2}(t) &= \frac{1}{2!}\sum_{j=0}^{2}(-1)^j C_3^j (t+2-j)^2 \\ &= \frac{1}{2!}\left[\frac{3!}{3!}(t+2)^2 - \frac{3!}{2!}(t+1)^2 + \frac{3!}{2!}t^2\right] = \frac{1}{2}(t-1)^2 \end{aligned}$$

$$F_{1,2}(t) = \frac{1}{2}(-2t^2+2t+1)$$

$$F_{2,2}(t) = \frac{1}{2}t^2$$

代入B样条曲线方程，得

$$P(t)=\frac{1}{2}(t-1)^2p_0+\frac{1}{2}(-2t^2+2t+1)p_1+\frac{1}{2}t^2p_2$$

按 t 的降幂整理：

$$P(t)=\frac{1}{2}[(p_0-2p_1+p_2)t^2+(-2p_0+2p_1)t+(p_0+p_1)]$$

写成矩阵形式为

$$P(t)=\frac{1}{2}[t^2\quad t\quad 1]\begin{bmatrix}1&-2&1\\-2&2&0\\1&1&0\end{bmatrix}\begin{bmatrix}p_0\\p_1\\p_2\end{bmatrix}$$

当 $t=0$ 时，$P(0)=\frac{1}{2}(p_0+p_1)$；当 $t=1$ 时，$P(1)=\frac{1}{2}(p_1+p_2)$。这说明曲线段的两端点是控制多边形首、末边的中点，如图8.3-1所示。

对曲线求导得

$$P'(t)=(t-1)p_0+(-2t+1)p_1+tp_2$$

当 $t=0$ 时，$P'(0)=p_1-p_0$；当 $t=1$ 时，$P'(1)=p_2-p_1$。这说明曲线段两端点的切向量是控制多边形的两个边向量，如图8.3-1所示。

如果 $m+n>2$，那么依次取三点，例如 $p_0p_1p_2$，$p_1p_2p_3$，$p_2p_3p_4$，……，都可以得到一段二次B样条曲线段，总和起来就得到多段的二次B样条曲线，而且每段之间 C^1 连续，如图8.3-2所示。

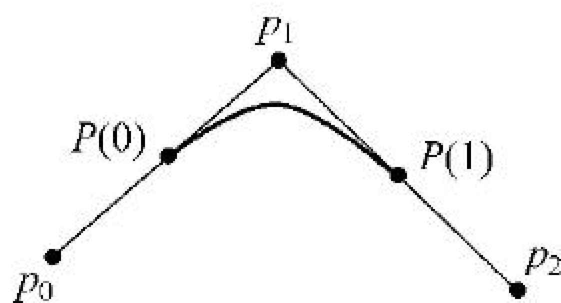


图8.3-1　二次B样条曲线

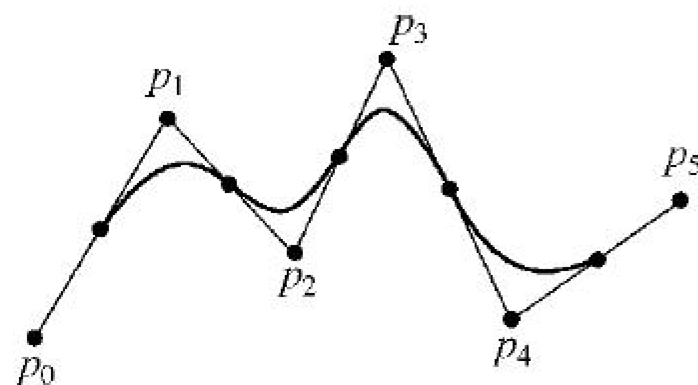


图8.3-2　多段的二次B样条曲线

对于三次B样条曲线(此时 $n=3$)，首先计算B样条基函数：

$$F_{0,3}(t)=\frac{1}{3!}\sum_{j=0}^{3}(-1)^jC_4^j(t+3-j)^3=\frac{1}{6}(-t^3+3t^2-3t+1)$$

$$F_{1,3}(t)=\frac{1}{6}(3t^3-6t^2+4)$$

$$F_{2,3}(t)=\frac{1}{6}(-3t^3+3t^2+3t+1)$$

$$F_{3,3}(t)=\frac{1}{6}t^3$$

代入B样条曲线方程，并整理得

$$P(t)=\frac{1}{6}[(-p_0+3p_1-3p_2+p_3)t^3+(3p_0-6p_1+3p_2)t^2+(-3p_0+3p_2)t+(p_0+4p_1+p_2)]$$

矩阵表达式为

$$P(t)=\frac{1}{6}[t^3\quad t^2\quad t\quad 1]\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 0 & 3 & 0\\ 1 & 4 & 1 & 0\end{bmatrix}\begin{bmatrix}p_0\\ p_1\\ p_2\\ p_3\end{bmatrix}$$

当 $t=0$ 时，

$$P(0)=\frac{1}{6}(p_0+4p_1+p_2)=\frac{1}{3}\left(\frac{p_0+p_2}{2}\right)+\frac{2}{3}p_1$$

当 $t=1$ 时，

$$P(1)=\frac{1}{6}(p_1+4p_2+p_3)=\frac{1}{3}\left(\frac{p_1+p_3}{2}\right)+\frac{2}{3}p_2$$

这说明三次 B 样条曲线的起点 $P(0)$ 落在 $\triangle p_0p_1p_2$ 的中线 $p_1p_1^*$ 上，离 p_1 1/3 处，终点 $P(1)$ 落在 $\triangle p_1p_2p_3$ 的中线 $p_2p_2^*$ 上，离 p_2 1/3 处。

对三次 B 样条曲线求导得

$$P'(t)=\frac{1}{6}[3(-p_0+3p_1-3p_2+p_3)t^2+2(3p_0-6p_1+3p_2)t+(-3p_0+3p_2)]$$

当 $t=0$ 时，$P'(0)=\frac{1}{2}(p_2-p_0)$；当 $t=1$ 时，$P'(1)=\frac{1}{2}(p_3-p_1)$。这说明三次 B 样条曲线段始点处的切向量 $P'(0)$ 平行于 $\triangle p_0p_1p_2$ 的底边 p_0p_2，其长度为底边长的一半。终点处的切向量 $P'(1)$ 平行于 $\triangle p_1p_2p_3$ 的底边 p_1p_3，其长度为它的一半。

对三次 B 样条曲线求二次导数得

$$P''(t)=\frac{1}{6}[6(-p_0+3p_1-3p_2+p_3)t+2(3p_0-6p_1+3p_2)]$$

当 $t=0$ 时，$P''(0)=(p_2-p_1)+(p_0-p_1)$；当 $t=1$，$P''(1)=(p_3-p_2)+(p_1-p_2)$。这说明三次 B 样条曲线段始点处的二阶导向量 $P''(0)$ 等于中线向量 $p_1p_1^*$ 的两倍，终点处的二阶导向量 $P''(1)$ 为中线向量 $p_2p_2^*$ 的两倍，如图 8.3-3 所示。

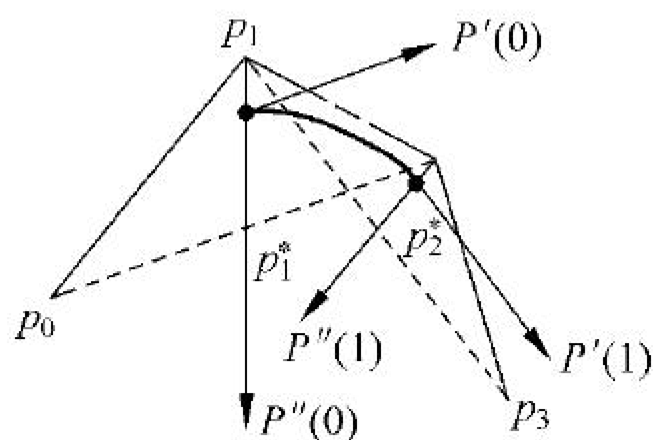


图 8.3-3 三次 B 样条曲线

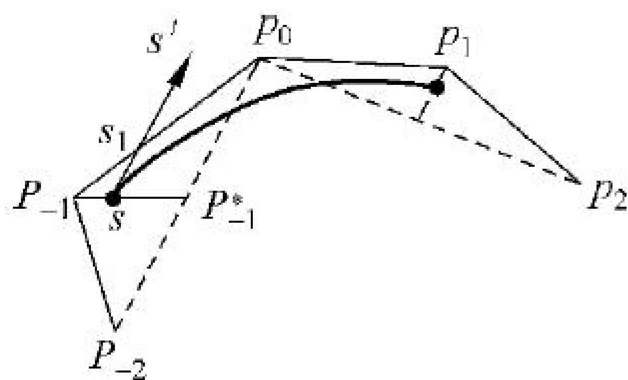


图 8.3-4 多段三次 B 样条曲线自动连续

如果给出了五个控制顶点 $p_i(i=0,1,2,3,4)$，前四个控制顶点(p_0,p_1,p_2,p_3)决定第一曲线段，而后四个控制顶点$(p_1、p_2、p_3、p_4)$决定第二曲线段。由于第一曲线段终点信息（位置、一阶、二阶导向量）和第二曲线段始点信息（位置、一阶、二阶导向量）都仅与 $\triangle p_1p_2p_3$ 有关，且都分别相等，这样前后两个线段在连接点处自动二阶连续。这说明三次 B 样条曲线的二阶连续性可以直接得到保证，而不必进行附加处理，因而在使用中非常方便，如图 8.3-4 所示。

8.3.3　双三次B样条曲面

将B样条曲线段拓广为B样条曲面块，其数学表达式为

$$P(u,w)=\sum_{i=0}^{3}\sum_{j=0}^{3}F_{i,3}(u)F_{j,3}(w)p_{ij},\quad 0\leqslant u\leqslant 1,0\leqslant w\leqslant 1$$

其矩阵表达式为

$$P(u,w)=\boldsymbol{U}\boldsymbol{M}_B\boldsymbol{P}\boldsymbol{M}_B^{\mathrm{T}}\boldsymbol{W}^{\mathrm{T}}$$

其中

$$\boldsymbol{M}_B=\frac{1}{6}\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 0 & 3 & 0\\ 1 & 4 & 1 & 0\end{bmatrix},\quad \boldsymbol{M}_B^{\mathrm{T}}=\frac{1}{6}\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 0 & 4\\ -3 & 3 & 3 & 1\\ 1 & 0 & 0 & 0\end{bmatrix}$$

$$\boldsymbol{W}^{\mathrm{T}}=\begin{bmatrix}w^3\\ w^2\\ w\\ 1\end{bmatrix},\quad \boldsymbol{P}=\begin{bmatrix}p_{00} & p_{01} & p_{02} & p_{03}\\ p_{10} & p_{11} & p_{12} & p_{13}\\ p_{20} & p_{21} & p_{22} & p_{23}\\ p_{30} & p_{31} & p_{32} & p_{33}\end{bmatrix},\quad \boldsymbol{U}=[u^3\quad u^2\quad u\quad 1]$$

$\boldsymbol{P}$是由各顶点p_{ij}组成的矩阵，依次用线段连接p_{ij}中的相邻两顶点构成的空间网格，称为B特征网格，B特征网格所决定的B样条曲面块一般情况下不通过B特征网格的任意一个顶点。特征网格及双三次B样条曲面如图8.3-5所示。

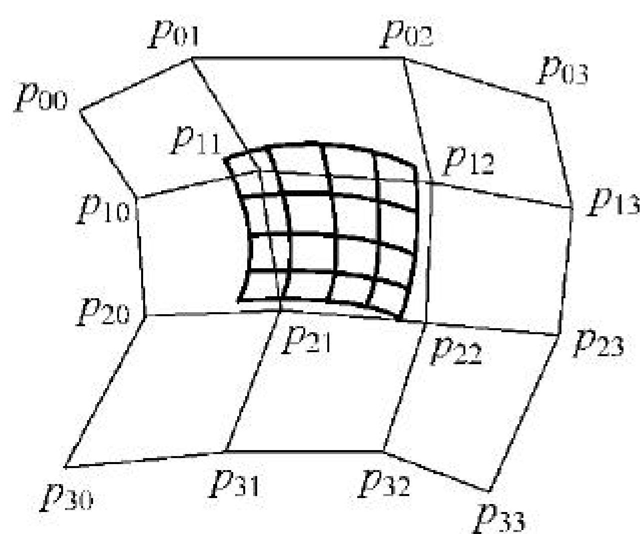


图8.3-5　双三次B样条曲面

和B样条曲线一样，双三次B样条曲面的优点是，自然地解决了曲面片之间的C^2连续连接问题，所以双三次B样条曲面很适合于进行外形设计。但是，由于双三次B样条曲面片一般情况下不通过B特征网格的任意一个顶点，因此B样条曲面不便于插值。

8.3.4　B样条递推定义

B样条除了一般定义外，还有de Boor-Cox递推定义。de Boor-Cox递推定义对B样条曲线不再按分段来划分控制顶点，参数t也不再在每一分段曲线上定义参数区间，而是在整个控制顶点序列中定义节点矢量。

B样条曲线方程定义为$P(t)=\sum_{k=0}^{n}p_kN_{k,m}(t)$，其中，$p_k(k=0,1,2,\cdots,n)$是控制多边形的顶点，$N_{k,m}(t)(k=0,1,2,\cdots,n)$称为$m$阶($m-1$次)B样条基函数，对于参数$t$不再局限于[0,1]区间，而是用一个节点矢量$\boldsymbol{T}$来表示，其中的节点值是非递减序列。$n+1$个控制顶点和$m$阶B样条节点矢量为

$$\boldsymbol{T}=(t_0,t_1,\cdots,t_{m-1},t_m,\cdots,t_n,t_{n+1},\cdots,t_{n+m-1},t_{n+m})$$

这时，B样条基函数可以用如下的递推公式表示：

$$N_{k,1}(t)=\begin{cases}1, & t_k \leqslant t < t_{k+1} \\ 0, & \text{其他}\end{cases}$$

$$N_{k,m}(t)=\frac{t-t_k}{t_{k+m-1}-t_k}N_{k,m-1}(t)+\frac{t_{k+m}-t}{t_{k+m}-t_{k+1}}N_{k+1,m-1}(t)$$

并约定$\frac{0}{0}=0$。

该递推公式表明：欲确定第k个m阶B样条$N_{k,m}(t)$，需要用到$t_k,t_{k+1},\cdots,t_{k+m}$共$m+1$个节点，称区间$[t_k,t_{k+m}]$为$N_{k,m}(t)$的支承区间，因此，B样条的基函数是一个分段函数。在区间$[t_k,t_{k+m}]$，$N_{k,m}(t)\neq 0$，在其他区间$N_{k,m}(t)=0$，这个特性称为局部性。B样条的局部性对曲线曲面的设计有两方面的影响：一是第k段曲线段在两个相邻节点值$[t_k,t_{k+1}]$上仅由m个控制点控制，若要修改该段曲线，仅修改这m个控制点即可；二是修改控制顶点p_k对B样条曲线的影响是局部的。对于均匀m次B样条曲线，调整一个顶点p_k的位置只影响B样条曲线在区间$[t_k,t_{k+m}]$的部分，即最多只影响与该顶点有关的m段曲线。所以，局部性质是B样条曲线最具魅力的性质。

8.3.5 B样条曲线的类型

B样条曲线按其节点矢量中节点的分布情况可划分为四种类型：均匀B样条曲线、准均匀B样条曲线、分段Bézier曲线和非均匀B样条曲线。

1. 均匀B样条曲线

节点矢量中节点为沿参数t轴均匀或等距分布，所有节点区间长度为常数，这样的节点矢量定义了均匀B样条基。比如：$\boldsymbol{T}=(-2,-1.5,-1,-0.5,0,0.5,1,1.5,2)$，$\boldsymbol{T}=(0,1,2,3,4,5,6,7)$，它为如图8.3-6所示的三次均匀B样条曲线。

2. 准均匀B样条曲线

它与均匀B样条曲线的差别在于两端节点具有重复度m，这样的节点矢量定义了准均匀的B样条基。均匀B样条曲线没有保留Bézier曲线端点的几何性质，即样条曲线的首末端点不再是控制多边形的首末端点。采用准均匀的B样条曲线解决了这个问题，如图8.3-7所示。

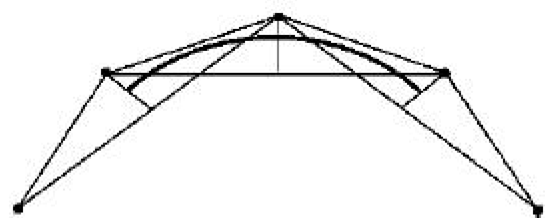


图8.3-6　均匀B样条曲线

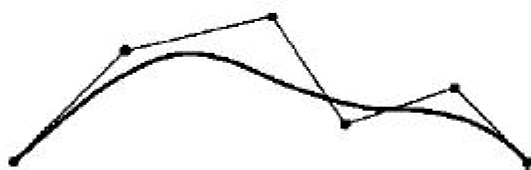


图8.3-7　准均匀B样条曲线

3. 分段Bézier曲线

节点矢量中两端节点具有重复度m，所有内节点的重复度为$m-1$，这样的节点矢量定义了分段的Bernstein基。B样条曲线用分段Bézier曲线表示后，各曲线段就具有了相对的独立性，移动曲线段内的一个控制顶点只影响该曲线段的形状，对其他曲线段的形状没有影响。并且Bézier曲线一整套简单有效的算法都可以原封不动地采用。其缺点是增加了定

义曲线的数据、控制顶点数及节点数。分段 Bézier 曲线的实例如图 8.3-8 所示。

4. 非均匀 B 样条曲线

在这种类型中，任意分布的节点矢量 $\boldsymbol{T}=(t_0,t_1,\cdots,t_{m-1},t_m,\cdots,t_n,t_{n+1},\cdots,t_{n+m-1},t_{n+m})$ 只要在数学上成立（节点序列非递减，两端节点重复度 $\leqslant k$，内节点重复度 $\leqslant k-1$）都可选取。这样的节点矢量定义了非均匀 B 样条基。

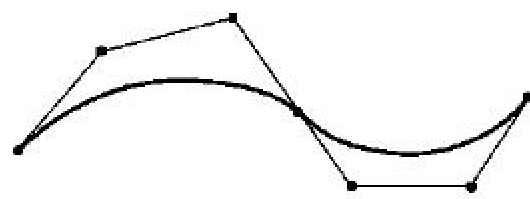


图 8.3-8 三次分段 Bézier 曲线

8.3.6 反求 B 样条曲线控制点

用分段三次 B 样条曲线来拟合，根据端点性质，其上型值点 q_i 和控制点 p_i 的位置矢量之间有以下关系：

$$q_i=\frac{1}{6}(p_{i-1}+4p_i+p_{i+1}),\quad i=1,2,\cdots,n$$

假定需求首末两点过 q_1 和 q_n 的准均匀的 B 样条曲线，则 $p_1=q_1$，$p_n=q_n$。于是求解控制顶点 $P_i(i=2,3,\cdots,n-1)$ 的线性方程组为

$$\begin{bmatrix}4&1&0&0&0&0&0&0\\1&4&1&0&0&0&0&0\\0&1&4&1&0&0&0&0\\0&0&.&.&.&0&0&0\\0&0&0&.&.&.&0&0\\0&0&0&0&.&.&.&0\\0&0&0&0&0&1&4&1\\0&0&0&0&0&0&1&4\end{bmatrix}\begin{bmatrix}p_2\\p_3\\.\\.\\.\\.\\p_{n-2}\\p_{n-1}\end{bmatrix}=\begin{bmatrix}6q_2-q_1\\6q_3\\.\\.\\.\\.\\6q_{n-2}\\6q_{n-1}-q_n\end{bmatrix}$$

补充两个边界条件为

$$p_0=p_{-1}=q_1,\quad p_{n+1}=p_{n+2}=q_n$$

8.3.7 B 样条曲线绘制

在应用程序中绘制曲线时，一种比较简单的方法是利用鼠标拾取一系列控制顶点，然后根据 B 样条的一般定义，计算 B 样条曲线上的点并连线即可。首先，要建立 B 样条曲线的数据结构，B 样条曲线的结构类可简单创建为（放在程序的 BasicClass.h 文件中）：

```
class CBSplineCurve{
public:
    CArray<CVector,CVector> m_Array_Vector;     //控制顶点
    CArray<CPoint,CPoint> m_Array_Point;        //原始控制多边形顶点
    CBSplineCurve& operator=(const CBSplineCurve &BSplineCurve){
        if(this == &BSplineCurve)
            return *this;
        else{
            m_Array_Vector.RemoveAll();
            m_Array_Vector.Append(BSplineCurve.m_Array_Vector);
```

```
                m_Array_Point.RemoveAll();
                m_Array_Point.Append(BSplineCurve.m_Array_Point);
                //Degree_Num = BSplineCurve.Degree_Num;
                return * this;
            }
        }
        CBSplineCurve(){};
        CBSplineCurve(const CBSplineCurve &BSplineCurve){
            m_Array_Vector.Append(BSplineCurve.m_Array_Vector);
            m_Array_Point.Append(BSplineCurve.m_Array_Point);
        };
        void Reset(){      //格式化B样条
            m_Array_Vector.RemoveAll();
            m_Array_Point.RemoveAll();
        }
        bool operator ==(CBSplineCurve &BSplineCurve){
            if(this == &BSplineCurve) return true;
            else
    if(this->m_Array_Vector.GetSize() == BSplineCurve.m_Array_Vector.GetSize()){
                for(int i = 0;i < this->m_Array_Vector.GetSize();i++){
    if((this->m_Array_Vector.GetAt(i) == BSplineCurve.m_Array_Vector.GetAt(i)) == false)
                        return false;
                }
                return true;
            }
            else
                return false;
        }
    };
```

在CGTest002View.h头文件中添加B样条曲线变量以及交互操作所需的其他相关变量：

```
    CArray<CBSplineCurve,CBSplineCurve> m_BSplineCurve_Array;    //B样条曲线集合
    CBSplineCurve TmpBSplineCurve;                         //临时B样条曲线,整条曲线还未最后确定
    CPoint B_Pt1,B_Pt2;                                    //交互时所需的点
    int BiStep;                                            //拾取步骤 0: 拾取第一个点,1: 拾取第二个点
```

在工具栏中设置绘制B样条的工具条标识ID,并加入其消息映射函数,在该函数中,设置绘制B样条的flag：m_iFlag=16,以及拾取控制顶点的步骤：BiStep=0。然后,用鼠标在显示屏幕上拾取控制顶点,为了方便观察,在绘制临时B样条曲线的同时也绘制通过控制顶点的控制多边形。拾取控制顶点的操作在OnLButtonDown()中实现：

```
    void CCGTest002View::OnLButtonDown(UINT nFlags, CPoint point) {
    ...//(原有代码此处省略)
    else if(this->m_iFlag == 16){      //m_iFlag = 16 标示绘制B样条曲线,获取控制点
            if(BiStep == 0){                                   //步骤一,先获取一个点
                this->B_Pt1 = this->B_Pt2 = point;
                CVector Vt;                                    //把拾取赋给Vt
                Vt.v_x = point.x;
```

```
            Vt.v_y = point.y;
            Vt.v_z = 0;
            TmpBSplineCurve.m_Array_Vector.Add(Vt);         //插入临时B样条点集合中
            TmpBSplineCurve.m_Array_Point.Add(point);
            BiStep = 1;
        }
        else if(BiStep == 1){                          //拾取第二个点,并生成控制多边形直线
            CDC * pDC = GetDC();
            DrawLine(pDC,this->B_Pt1,point);
            ReleaseDC(pDC);
            CVector Vt;
            Vt.v_x = point.x;
            Vt.v_y = point.y;
            Vt.v_z = 0;
            TmpBSplineCurve.m_Array_Vector.Add(Vt);         //插入临时B样条点集合中
            TmpBSplineCurve.m_Array_Point.Add(point);
            this->B_Pt1 = this->B_Pt2 = point;
            BiStep = 1;
            Invalidate();
        }
    }
}
```

在移动鼠标时,采用橡皮筋技术动态绘制控制多边形:

```
void CCGTest002View::OnMouseMove(UINT nFlags, CPoint point) {
...//(原有代码此处省略)
else if(this->m_iFlag == 16){
        if(BiStep == 1){                                //画线
            CDC * pDC = GetDC();
            pDC->SetROP2(R2_NOT);                       //XORPEN
            DrawLine(pDC,B_Pt1,B_Pt2);
            DrawLine(pDC,B_Pt1,point);
            B_Pt2 = point;
            ReleaseDC(pDC);
        }
    }
}
```

当右击时,完全确定B样条曲线控制顶点:

```
void CCGTest002View::OnRButtonDown(UINT nFlags, CPoint point) {
...//(原有代码此处省略)
    if(m_iFlag == 16) {
        //把临时曲线加入B样条曲线集合中,点数超过3个才加入
        if(TmpBSplineCurve.m_Array_Vector.GetSize()> 3)
            m_BSplineCurve_Array.Add(TmpBSplineCurve);
        TmpBSplineCurve.Reset();                        //格式化临时B样条
    }
}
```

临时B样条曲线和最后确定的B样条曲线都在OnDraw()中调用的DrawG()中绘制:

```
void CCGTest002View::DrawG(CDC * pDC){
...//(原有代码此处省略)
    //画曲线
    if(this->m_BSplineCurve_Array.GetSize()>0){            //逐条绘制
        CBSplineCurve BCurve;
        for(int i=0;i<m_BSplineCurve_Array.GetSize();i++){
            BCurve=m_BSplineCurve_Array.GetAt(i);
//if(pick_bcurve==m_Picker.picktype&&BCurve==m_Picker.m_BSplineCurve)
//                continue;     //曲线被拾取时,此处不绘制,暂不使用
            if(BCurve.m_Array_Vector.GetSize()>3) {        //三次B样条
    DrawBSplineCurve(pDC,BCurve.m_Array_Vector,m_DrawColor,m_LineWidth);
            }
        }
    }
    if(TmpBSplineCurve.m_Array_Point.GetSize()>1){          //绘制临时B样条曲线
        //既画控制多边形,也绘制曲线
        for(int i=1;i<TmpBSplineCurve.m_Array_Point.GetSize();i++){
DrawLine(pDC,TmpBSplineCurve.m_Array_Point.GetAt(i-1),TmpBSplineCurve.m_Array_Point.GetAt(i));
        }
        //当控制顶点多于4个时,画样条
        if(TmpBSplineCurve.m_Array_Vector.GetSize()>3){   //画样条曲线
DrawBSplineCurve(pDC,TmpBSplineCurve.m_Array_Vector,m_DrawColor,m_LineWidth);
        }
    }
}
```

其中,调用绘制B样条曲线的函数DrawBSplineCurve()放在BasicGraph1.h文件中:

```
/*********************************************************************
    DrawBSplineCurve: 绘制三次B样条曲线
pDC: 显示器设备指针; m_Array_Vector: 控制顶点; crColor: 曲线颜色; lineWidth: 线宽
    *******************************************************************/
    void DrawBSplineCurve(CDC * pDC, CArray<CVector,CVector> &m_Array_Vector, COLORREF
crColor, int lineWidth=0){
        if(m_Array_Vector.GetSize()<=3)return;             //控制顶点少于4个,不绘制
        CVector Vt0,Vt1;
        double t=0;                                        //参数
        //计算第一个曲线点
    Vt0=GetBSplineCurvePt(t,m_Array_Vector.GetAt(0),m_Array_Vector.GetAt(1),m_Array_
Vector.GetAt(2),m_Array_Vector.GetAt(3));
        int rate=100;                                      //设置曲线分段数为100
        for(int i=0;i<m_Array_Vector.GetSize()-3;i++){
            for(t=0;t<1;t+=1.0/rate){                      //计算第二个曲线点
    Vt1=GetBSplineCurvePt(t,m_Array_Vector.GetAt(i),m_Array_Vector.GetAt(i+1),m_Array_
Vector.GetAt(i+2),m_Array_Vector.GetAt(i+3));
                //在屏幕投影连线
    MIDPOINT_Line(pDC,CPoint((int)(Vt0.v_x+0.5),(int)(Vt0.v_y+0.5)),CPoint((int)(Vt1.
v_x+0.5),(int)(Vt1.v_y+0.5)),crColor,lineWidth);
                Vt0=Vt1;
            }
        }
    }
```

其中，计算曲线点的函数代码为：

```
CVector GetBSplineCurvePt(double t,CVector Pt0,CVector Pt1,CVector Pt2,CVector Pt3){
    double F03,F13,F23,F33;
    CVector Pt;
    F03 = (-t*t*t+3*t*t-3*t+1)/6;                        //计算 B 样条基函数
    F13 = (3*t*t*t-6*t*t+4)/6;
    F23 = (-3*t*t*t+3*t*t+3*t+1)/6;
    F33 = t*t*t/6;
    Pt.v_x = Pt0.v_x*F03+Pt1.v_x*F13+Pt2.v_x*F23+Pt3.v_x*F33;   //计算曲线点的 x 值
    Pt.v_y = Pt0.v_y*F03+Pt1.v_y*F13+Pt2.v_y*F23+Pt3.v_y*F33;   //计算曲线点的 y 值
    return Pt;
}
```

图 8.3-9 所示为利用上述代码实现的拾取控制顶点绘制控制多边形和三次 B 样条曲线的效果。

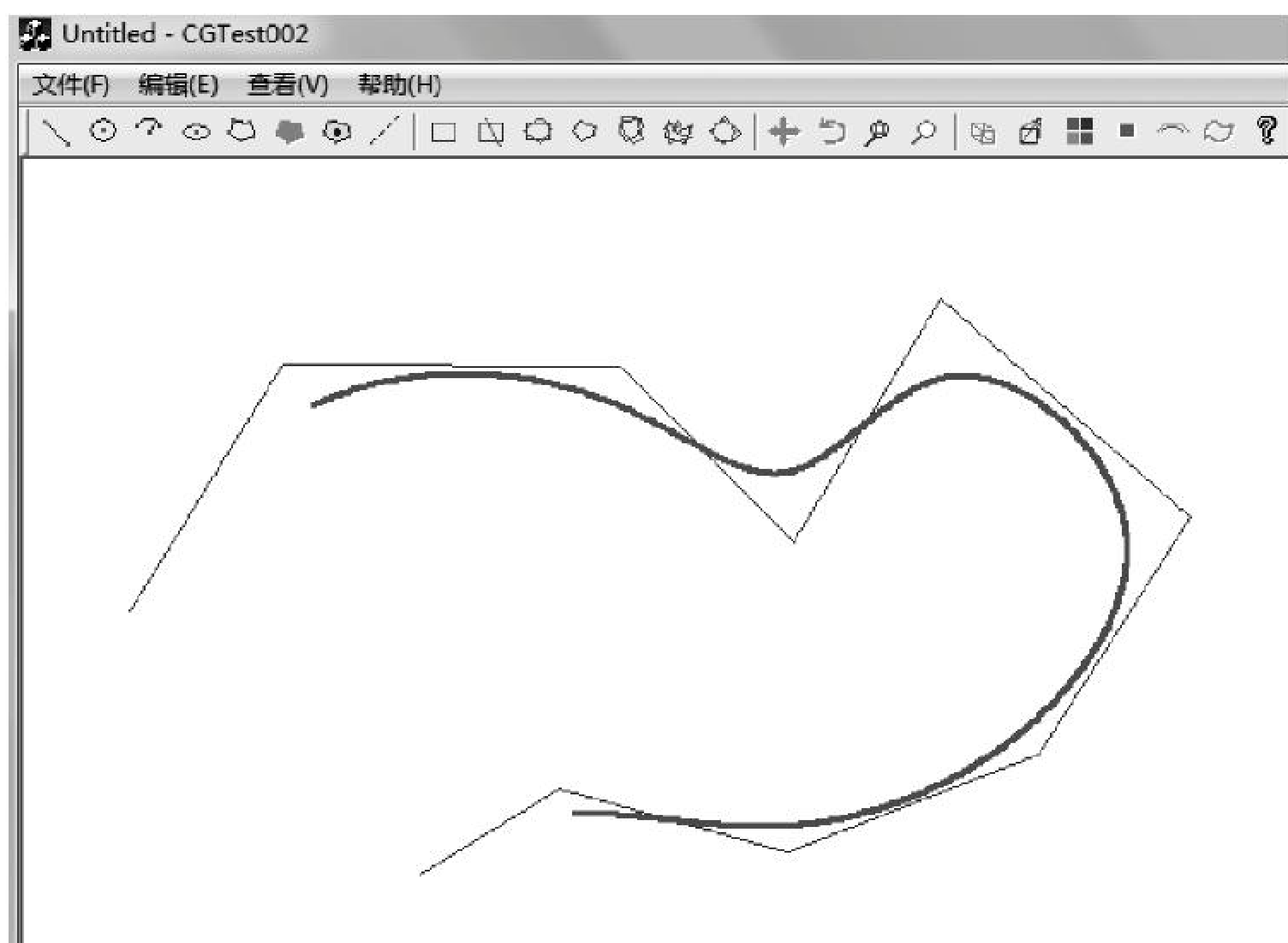


图 8.3-9　B 样条曲线及控制多边形

在对曲线进行操作时，例如图形变换等，就需要使用拾取交互技术实现对曲线的拾取操作。首先，在拾取的枚举中加入曲线的类型 pick_bcurve：

```
enum Picktype { …,pick_bcurve};
```

在拾取类中加入 B 样条曲线的变量：

```
class CPicker:CDraw{
    … //(原有代码此处省略)
    CBSplineCurve m_BSplineCurve;
};
```

在鼠标移动消息函数中，增加判断鼠标是否拾取到 B 样条曲线的代码：

```
void CCGTest002View::OnMouseMove(UINT nFlags, CPoint point) {
```

```
    if(this->m_iFlag==-1){
    ... //(原有代码此处省略)
        else if(CheckIsPicked(point,this->m_BSplineCurve_Array,m_Picker0)==true)
        {//是否在B样条曲线上
            iflag=1;
        }
        else
            iflag=0;
    }
}
```

其中，曲线拾取判断函数 CheckIsPicked()和其他图形拾取判断函数一样写在 Basicgraph1.h 文件中：

```
bool CheckIsPicked(CPoint &point, CArray<CBSplineCurve, CBSplineCurve> &m_BSplineCurve_
Array, CPicker &m_Picker) {
        CBSplineCurve BSplineCurve;
        for(int i=0;i<m_BSplineCurve_Array.GetSize();i++){
            BSplineCurve=m_BSplineCurve_Array.GetAt(i);
            //逐条判断是否在B样条曲线上
            if(CheckIsPicked(point,BSplineCurve,m_Picker)==true)
                return true;
        }
        return false;
}
```

由于B样条曲线是分段曲线，所以，在判断是否B样条曲线时，按分段分别判断，首先对每段曲线的控制多边形构成的包围盒进行粗略判断；然后，判断屏幕拾取点是否在曲线的投影点上。代码如下：

```
bool CheckIsPicked(CPoint &point,CBSplineCurve m_BSplineCurve,CPicker &m_Picker){
    for(int i=0;i<m_BSplineCurve.m_Array_Vector.GetSize()-3;i++){
        int x1,x2,y1,y2;
        x1 = x2 = m_BSplineCurve.m_Array_Vector.GetAt(i).v_x;
        y1 = y2 = m_BSplineCurve.m_Array_Vector.GetAt(i).v_y;
        for(int k=i;k<i+3;k++){
            if(x1>m_BSplineCurve.m_Array_Vector.GetAt(k).v_x)
                x1 = m_BSplineCurve.m_Array_Vector.GetAt(k).v_x;
            else if(x2<m_BSplineCurve.m_Array_Vector.GetAt(k).v_x)
                x2 = m_BSplineCurve.m_Array_Vector.GetAt(k).v_x;

            if(y1>m_BSplineCurve.m_Array_Vector.GetAt(k).v_y)
                y1 = m_BSplineCurve.m_Array_Vector.GetAt(k).v_y;
            else if(y2<m_BSplineCurve.m_Array_Vector.GetAt(k).v_y)
                y2 = m_BSplineCurve.m_Array_Vector.GetAt(k).v_y;
        }
        //首先判断是否在控制多边形构成的包围盒内,如在,则继续判断
        if(CheckIsInBox(point,x1,x2,y1,y2)==false)
            continue;
        int rate=100;                                      //设置曲线分段数为100
        CVector Vt;
```

```
        for(double t = 0;t < 1;t += 1.0/rate){                //计算曲线上的点
    Vt = GetBSplineCurvePt(t,m_BSplineCurve.m_Array_Vector.GetAt(i),m_BSplineCurve.m_Array_
    Vector.GetAt(i + 1),m_BSplineCurve.m_Array_Vector.GetAt(i + 2),m_BSplineCurve.m_Array_
    Vector.GetAt(i + 3));
        //判断鼠标点与曲线投影点的距离是否在精度内(例如距离的平方小于81)
            if((Vt.v_x - point.x) * (Vt.v_x - point.x) + (Vt.v_y - point.y) * (Vt.v_y - point.
    y)< 81){
                m_Picker.picktype = pick_bcurve;              //拾取到曲线
                m_Picker.m_BSplineCurve = m_BSplineCurve;
                return true;
            }
        }
    }
    return false;
}
```

确定拾取曲线，在CopyPicker()函数中将拾取的样条曲线复制给进行图形操作的拾取变量m_Picker，代码如下：

```
void CopyPicker(CPicker &m_Picker,CPicker &m_Picker0){
    ...//(原有代码此处省略,下面的代码加在实体复制代码后面)
    else if(m_Picker.picktype == pick_bcurve){               //6.判断是否样条
        //复制样条
        m_Picker.m_BSplineCurve = m_Picker0.m_BSplineCurve;
    }
}
```

同理，拾取图形的绘制函数DrawPicker中也需加入绘制拾取的B样条曲线的代码：

```
void DrawPicker(CDC * pDC,CPicker &m_Picker,COLORREF crColor,double m_Matrix_V[][4],CBody_
Stretch Body[],int bodyNum = 0,int lineWidth = 0,bool hideFlag = false){
...//(原有代码此处省略,下面的代码加在绘制拾取的实体后面)
    else if(m_Picker.picktype == pick_bcurve)                //7. 绘制拾取的B样条曲线
DrawBSplineCurve(pDC,m_Picker.m_BSplineCurve.m_Array_Vector,crColor,lineWidth);
}
```

由于B样条曲线具有几何不变性，因此，通过对B样条曲线控制顶点进行图形变换即可实现对B样条曲线的图形变换。在拾取图形的图形变换函数中加入B样条曲线图形变换的相关代码：

```
void TransforOf_2D_Picker(CPicker& m_Picker,double m_Matrix[][3]){
    ...//(原有代码此处省略)
    else if(m_Picker.picktype == pick_bcurve){               //对每个控制顶点进行变换
        CArray < CVector,CVector > bcurve;
        CVector Vt;
        CPoint Pt;
        for(int i = 0;i < m_Picker.m_BSplineCurve.m_Array_Vector.GetSize();i++){
            Pt.x = (int)(m_Picker.m_BSplineCurve.m_Array_Vector.GetAt(i).v_x + 0.5);
            Pt.y = (int)(m_Picker.m_BSplineCurve.m_Array_Vector.GetAt(i).v_y + 0.5);
            GetNewPoint(Pt,m_Matrix);
            Vt.v_x = Pt.x;
```

```
                Vt.v_y = Pt.y;
                bcurve.Add(Vt);
            }
            m_Picker.m_BSplineCurve.m_Array_Vector.RemoveAll();
            m_Picker.m_BSplineCurve.m_Array_Vector.Append(bcurve);
        }
    }
```

8.3.8 曲面拉伸造型方法

自由曲面造型是三维图形造型的一个重要研究内容。除了 Bézier 曲面和双三次 B 样条曲面生成方法外，对曲线进行拉伸、扫描、旋转以及放样等也可实现曲面造型，而且这些造型方法能够满足造型要求，操作方便，所以在工程上广泛使用。

B 样条曲面拉伸造型方法的思路是：将一平面 B 样条曲线沿着和该平面垂直的方向拉伸一定的距离形成曲面。这种拉伸曲面的特点是：当与一个和原始平面平行的平面截交时，交线是一个和原始曲线等距的 B 样条曲线。

B 样条拉伸曲面的数据结构可创建如下：

```
class CBSplineSurf_Stretch{
public:
    CBSplineCurve BSplineCurve0,BSplineCurve1;      //拉伸的B样条曲线及拉伸后的曲线
    double Length;                                   //拉伸长度
    CBSplineSurf_Stretch(){
        Length = 0;
    };
    ~CBSplineSurf_Stretch(){};
    CBSplineSurf_Stretch(const CBSplineSurf_Stretch &BSplineSurfOfStretch){
        BSplineCurve0 = BSplineSurfOfStretch.BSplineCurve0;
        BSplineCurve1 = BSplineSurfOfStretch.BSplineCurve1;
        Length = BSplineSurfOfStretch.Length;
    };
    CBSplineSurf_Stretch& operator = (const CBSplineSurf_Stretch &BSplineSurfOfStretch){
        if(this == &BSplineSurfOfStretch)
            return *this;
        else{
            BSplineCurve0 = BSplineSurfOfStretch.BSplineCurve0;
            BSplineCurve1 = BSplineSurfOfStretch.BSplineCurve1;
            Length = BSplineSurfOfStretch.Length;
            return *this;
        }
    }
    bool operator == (CBSplineSurf_Stretch &BSplineSurfOfStretch){
        if(this == &BSplineSurfOfStretch) return true;
        else
        {//根据拉伸长度值以及初始控制多边形来判断两个曲面是否相同
            if(this->Length!= BSplineSurfOfStretch.Length)return false;
    for(int i = 0;i < BSplineSurfOfStretch.BSplineCurve0.m_Array_Point.GetSize();i++)
if(BSplineSurfOfStretch.BSplineCurve0.m_Array_Point.GetAt(i)!= this->BSplineCurve0.m_
```

```
Array_Point.GetAt(i))
                return false;
            return true;
        }
    }
    void Reset(){                                       //格式化
        Length = 0;
    };
};
```

在程序中创建拉伸曲面时，首先在CGTest002View.h头文件中添加曲面变量以及交互操作所需的其他相关变量：

```
CArray<CBSplineSurf_Stretch,CBSplineSurf_Stretch> m_BSplineSurf_Stretch_Array;
                                                    //曲面集合变量
CBSplineSurf_Stretch m_BSurf_Tmp;                   //临时B样条曲面
CBSurf_StretchDlg * m_BSurf_StretchDlg;             //曲面拉伸对话框
void CreateTmpStretchBSurf(double& m_dblLength);    //创建临时曲面
void CreateBSurfforStretch(double& m_dblLength);    //创建曲面
```

在设置曲面的拉伸长度时，可以创建一个非模式对话框如CBSurf_StretchDlg来完成。其中，调整拉伸长度的代码为：

```
void CBSurf_StretchDlg::OnDeltaposSpin1(NMHDR* pNMHDR, LRESULT* pResult) {
    NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
    UpdateData(TRUE);
    this->m_dblLength -= 1 * pNMUpDown->iDelta;
    this->m_pView->CreateTmpStretchBSurf(this->m_dblLength);
    UpdateData(FALSE);
    *pResult = 0;
}
void CBSurf_StretchDlg::OnOK() {
    UpdateData(TRUE);
    this->m_pView->CreateBSurfforStretch(this->m_dblLength);
    UpdateData(FALSE);
    CDialog::OnOK();
}
```

在CGTest002View类中创建临时曲面和最终曲面：

```
void CCGTest002View::CreateTmpStretchBSurf(double& m_dblLength){    //创建临时曲面
    if(this->m_Picker.picktype == pick_bcurve){     //对拾取的曲线拉伸实体
        if(CreateBSurfOfStretch(m_BSurf_Tmp,m_Picker,m_dblLength) == true)
            Invalidate();
    }
}
void CCGTest002View:: CreateBSurfforStretch(double& m_dblLength){   //创建最终曲面
    if(this->m_Picker.picktype == pick_bcurve){
        if(CreateBSurfOfStretch(m_BSurf_Tmp,m_Picker,m_dblLength) == true){
            CBSplineCurve BCurve;               //从曲线集合中去掉形成曲面的曲线
            for(int i = 0;i < m_BSplineCurve_Array.GetSize();i++){
                BCurve = m_BSplineCurve_Array.GetAt(i);
```

```
                if(pick_bcurve == m_Picker.picktype&&BCurve == m_Picker.m_BSplineCurve){
                    m_BSplineCurve_Array.RemoveAt(i);
                    break;
                }
            }
            this->m_BSplineSurf_Stretch_Array.Add(m_BSurf_Tmp);
            m_BSurf_Tmp.Reset();                //删除临时曲面
            this->m_Picker.picktype = pick_none;
            Invalidate();
        }
    }
}
```

其中,创建曲面的函数代码放在 BasicGraph1.h 文件中:

```
bool CreateBSurfOfStretch(CBSplineSurf_Stretch &m_BSurf, CPicker& m_Picker, double &m_
dblLength){
    if(m_Picker.picktype == pick_bcurve){
        //判断控制多边形的方向
        CArray<CPoint,CPoint> m_Array_Point;
        m_Array_Point.Append(m_Picker.m_BSplineCurve.m_Array_Point);
        int directionFlag = CheckDirofPolyline(m_Array_Point);      //判断方向
        m_BSurf.BSplineCurve0 = m_Picker.m_BSplineCurve;
        CBSplineCurve BSplineCurve;
        CArray<CVector,CVector> m_Array_Vector;                  ///控制顶点
        CVector Vt;
        if(directionFlag == 0){//顺时针方向
            for(int i = 0;i<m_Picker.m_BSplineCurve.m_Array_Vector.GetSize();i++){
                Vt = m_Picker.m_BSplineCurve.m_Array_Vector.GetAt(i);
                Vt.v_z = (-1) * m_dblLength;
                m_Array_Vector.Add(Vt);
            }
        }
        else{//逆时针方向
            for(int i = 0;i<m_Picker.m_BSplineCurve.m_Array_Vector.GetSize();i++)
            {
                Vt = m_Picker.m_BSplineCurve.m_Array_Vector.GetAt(i);
                Vt.v_z = m_dblLength;
                m_Array_Vector.Add(Vt);
            }
        }
        BSplineCurve.m_Array_Vector.Append(m_Array_Vector);BSplineCurve.m_Array_Point.
Append(m_Picker.m_BSplineCurve.m_Array_Point);
        m_BSurf.BSplineCurve1 = BSplineCurve;
        m_BSurf.Length = m_dblLength;
        //为了显示,将实体绕第一个点沿 x 轴旋转 30°,再沿 y 轴旋转 30°
    double m_Matrix[4][4],m_Matrix0[4][4];
        //首先移动到原点
        CPoint3D pt3D;
        Vt = m_BSurf.BSplineCurve0.m_Array_Vector.GetAt(0);
        pt3D.x = Vt.v_x;pt3D.y = Vt.v_y;pt3D.z = Vt.v_z;
```

```
        GetMatrix(m_Matrix,0,pt3D.x*(-1),pt3D.y*(-1),pt3D.z*(-1),0,1);
        GetMatrix(m_Matrix0,1,0,0,0,-30,1);                //沿 x 轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);                 //矩阵级联
        GetMatrix(m_Matrix0,2,0,0,0,-30,1);                //沿 y 轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);                 //矩阵级联
        GetMatrix(m_Matrix0,0,pt3D.x,pt3D.y,pt3D.z,0,1);  //移回原位置
        MatrixXMatrix(m_Matrix,m_Matrix0);                 //矩阵级联
        GetNewPoint(m_BSurf,m_Matrix);                     //曲面乘以变换矩阵,得新点
        return true;
    }
    else
        return false;
}
```

其中,判断控制多边形方向的函数为:

```
int CheckDirofPolyline(CArray<CPoint,CPoint> &m_point_Array0){//0: 顺时针,1: 逆时针
    CArray<CPoint,CPoint> m_point_Array;
    //1.判断多边形方向
    CPoint pt0,pt1,pt2;
    pt1 = m_point_Array0.GetAt(0);
    //查找极值
    int i_edge = 0;
    for(int i = 1;i < m_point_Array0.GetSize();i++){
        if(m_point_Array0.GetAt(i).y > pt1.y){
            pt1 = m_point_Array0.GetAt(i);
            i_edge = i;
        }
    }
    if(i_edge == 0){
        pt0 = m_point_Array0.GetAt(m_point_Array0.GetSize() - 1);
        pt2 = m_point_Array0.GetAt(1);
    }
    else if(i_edge == m_point_Array0.GetSize() - 1){
        pt0 = m_point_Array0.GetAt(m_point_Array0.GetSize() - 2);
        pt2 = m_point_Array0.GetAt(0);
    }
    else{
        pt0 = m_point_Array0.GetAt(i_edge - 1);
        pt2 = m_point_Array0.GetAt(i_edge);
    }
    m_point_Array.RemoveAll();
    if(VectorXVector(pt0,pt1,pt2)< 0)                      //多边形初始是顺时针
        return 0;
    else//> 0 表示多边形初始是逆时针方向
        return 1;
}
```

拉伸的曲面与几何变换矩阵相乘生成新点的函数为:

```
void GetNewPoint(CBSplineSurf_Stretch &m_BSurf,double m_Matrix[][4]){
    //m_BSurf 每个控制顶点变换
    CBSplineCurve BSplineCurve;
    CArray<CVector,CVector> bcurve;
    CVector Vt;
    CPoint3D Pt;
    BSplineCurve = m_BSurf.BSplineCurve0;
    for(int i = 0;i < BSplineCurve.m_Array_Vector.GetSize();i++){
        Vt = BSplineCurve.m_Array_Vector.GetAt(i);
        Pt.x = Vt.v_x;Pt.y = Vt.v_y;Pt.z = Vt.v_z;
        GetNewPoint(Pt,m_Matrix);
        Vt.v_x = Pt.x;Vt.v_y = Pt.y;Vt.v_z = Pt.z;
        bcurve.Add(Vt);
    }
    m_BSurf.BSplineCurve0.m_Array_Vector.RemoveAll();
    m_BSurf.BSplineCurve0.m_Array_Vector.Append(bcurve);
    BSplineCurve = m_BSurf.BSplineCurve1;
    bcurve.RemoveAll();
    for(i = 0;i < BSplineCurve.m_Array_Vector.GetSize();i++){
        Vt = BSplineCurve.m_Array_Vector.GetAt(i);
        Pt.x = Vt.v_x;Pt.y = Vt.v_y;Pt.z = Vt.v_z;
        GetNewPoint(Pt,m_Matrix);
        Vt.v_x = Pt.x;Vt.v_y = Pt.y;Vt.v_z = Pt.z;
        bcurve.Add(Vt);
    }
    m_BSurf.BSplineCurve1.m_Array_Vector.RemoveAll();
    m_BSurf.BSplineCurve1.m_Array_Vector.Append(bcurve);
}
```

曲面显示在 CCGTest002View 类的 OnDraw()的 DrawG()中调用：

```
void CCGTest002View::DrawG(CDC * pDC){
    …//(原有代码此处省略)
    //画曲面
    if(this->m_BSplineSurf_Stretch_Array.GetSize()>0){
        CBSplineSurf_Stretch BSplineSurf_Stretch;
        for(int i = 0;i < this->m_BSplineSurf_Stretch_Array.GetSize();i++){
    BSplineSurf_Stretch = this->m_BSplineSurf_Stretch_Array.GetAt(i);
if(BSplineSurf_Stretch == m_Picker.m_BSplineSurf_Stretch&&m_Picker.picktype == pick_bsurf)
    //拾取的曲面不画
                continue;
DrawBSplineSurf(pDC,BSplineSurf_Stretch,m_DrawColor,m_LineWidth);
        }
    }
    if(abs(this->m_BSurf_Tmp.Length)>0)
DrawBSplineSurf(pDC,this->m_BSurf_Tmp,m_DrawColor,m_LineWidth);
}
```

其中，详细绘制曲面的函数为(放在 BasicGraph1.h 文件中)：

```
void DrawBSplineSurf(CDC * pDC,CBSplineSurf_Stretch& BSplineSurf_Stretch, COLORREF crColor,
int lineWidth = 0){
    CVector Vt0,Vt1,Vt2,Vt3;
    int i,j;
    double t;
    //在 BSplineSurf_Stretch.BSplineCurve0 之间再画两条 B 样条线
    CArray<CVector,CVector> m_Array_Vector1,m_Array_Vector2;
    for(i = 0;i < BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetSize();i++){
        Vt0 = BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(i);
        Vt1 = BSplineSurf_Stretch.BSplineCurve1.m_Array_Vector.GetAt(i);
        Vt2.v_x = Vt0.v_x + (Vt1.v_x - Vt0.v_x)/3.0;
        Vt2.v_y = Vt0.v_y + (Vt1.v_y - Vt0.v_y)/3.0;
        Vt2.v_z = Vt0.v_z + (Vt1.v_z - Vt0.v_z)/3.0;
        Vt3.v_x = Vt0.v_x + (Vt1.v_x - Vt0.v_x) * 2/3.0;
        Vt3.v_y = Vt0.v_y + (Vt1.v_y - Vt0.v_y) * 2/3.0;
        Vt3.v_z = Vt0.v_z + (Vt1.v_z - Vt0.v_z) * 2/3.0;
        m_Array_Vector1.Add(Vt2);
        m_Array_Vector2.Add(Vt3);
    }//构造控制顶点
    //在拉伸的第一个和最后一个曲线的每个曲线段的 t = 0 及 t = 0.333,0.666 处连直线
    for(i = 0;i < BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetSize() - 3;i++){
        for(j = 0;j < 4;j++){
if(j == 3&&i!= BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetSize() - 4)
                continue;
            t = j * 1.0/3.0;
Vt0 = GetBSplineCurvePt (t, BSplineSurf _ Stretch. BSplineCurve0. m _ Array _ Vector. GetAt ( i),
BSplineSurf_ Stretch. BSplineCurve0. m _ Array _ Vector. GetAt ( i + 1), BSplineSurf _ Stretch.
BSplineCurve0.m _ Array _ Vector. GetAt ( i + 2), BSplineSurf _ Stretch. BSplineCurve0. m _ Array _
Vector.GetAt(i + 3));
Vt1 = GetBSplineCurvePt (t, BSplineSurf _ Stretch. BSplineCurve1. m _ Array _ Vector. GetAt ( i),
BSplineSurf_ Stretch. BSplineCurve1. m _ Array _ Vector. GetAt ( i + 1), BSplineSurf _ Stretch.
BSplineCurve1.m _ Array _ Vector. GetAt ( i + 2), BSplineSurf _ Stretch. BSplineCurve1. m _ Array _
Vector.GetAt(i + 3));
    MIDPOINT_Line(pDC,CPoint((int)(Vt0.v_x + 0.5),(int)(Vt0.v_y + 0.5)),CPoint((int)(Vt1.
v_x + 0.5),(int)(Vt1.v_y + 0.5)),crColor,lineWidth);
        }
    }
    //绘制四条 B 样条曲线
     DrawBSplineCurve (pDC, BSplineSurf _ Stretch. BSplineCurve0. m _ Array _ Vector, crColor,
lineWidth);
     DrawBSplineCurve (pDC, BSplineSurf _ Stretch. BSplineCurve1. m _ Array _ Vector, crColor,
lineWidth);
    DrawBSplineCurve(pDC,m_Array_Vector1,crColor,lineWidth);
    DrawBSplineCurve(pDC,m_Array_Vector2,crColor,lineWidth);
}
```

利用上述代码绘制的 B 样条拉伸曲面如图 8.3-10 所示。

与对曲线的图形操作类似,对曲面进行图形操作首先也需交互选取曲面,因此,在拾取

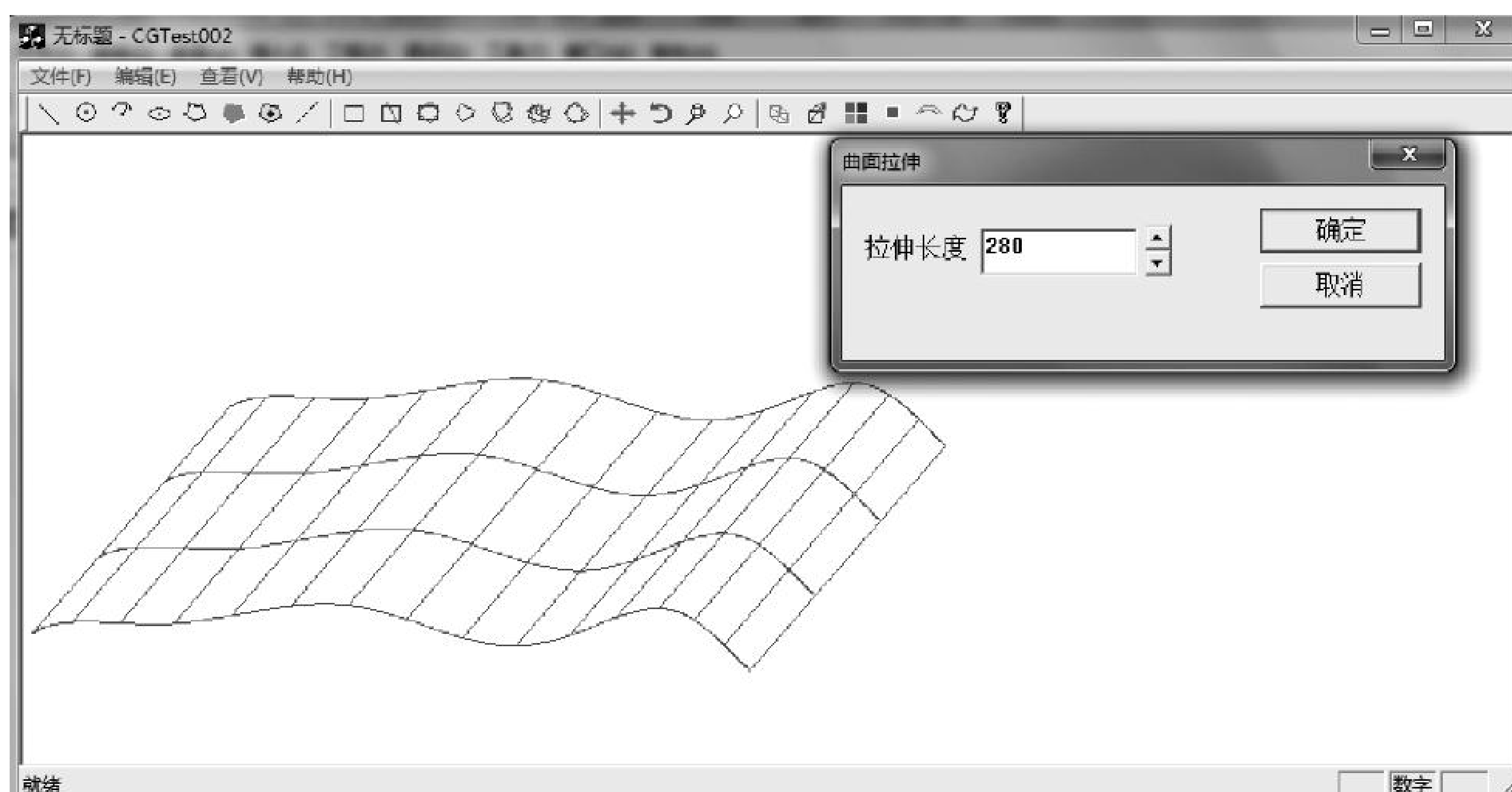


图 8.3-10　B 样条拉伸曲面

枚举结构中，加入曲面类型：

```
enum Picktype { …,pick_bsurf};
```

在拾取结构类中加入曲面变量：

```
class CPicker:CDraw{
…//(原有代码此处省略)
    CBSplineSurf_Stretch m_BSplineSurf_Stretch;
};
```

在鼠标移动消息函数 OnMouseMove()中，增加判断鼠标是否拾取到曲面的代码：

```
void CCGTest002View::OnMouseMove(UINT nFlags, CPoint point) {
    if(this->m_iFlag == -1){
    … //(原有代码此处省略)
        else    //是否在 B 样条曲面上
if(CheckIsPicked(point,this->m_BSplineSurf_Stretch_Array,m_Picker0) == true)
    iflag = 1;
    else
        iflag = 0;
    }
}
```

其中的判断函数 CheckIsPicked()写在 Basicgraph1.h 文件中，对于判断鼠标点是否拾取到曲面，本章采用的是一个简便方法：只判断鼠标点是否拾取了显示曲面的四个 B 样条曲线或者拉伸直线，如是，则认为拾取到曲面。函数代码如下：

```
bool CheckIsPicked(CPoint &point, CArray < CBSplineSurf_Stretch, CBSplineSurf_Stretch > &m_
BSplineSurf_Stretch_Array,CPicker &m_Picker){
    if(m_BSplineSurf_Stretch_Array.GetSize()> 0){
        CBSplineSurf_Stretch BSplineSurf_Stretch;
```

```
        double t;
        CVector Vt0,Vt1,Vt2,Vt3;
        int i,j,k;
        CLine line;
        for(i = 0;i < m_BSplineSurf_Stretch_Array.GetSize();i++){
            BSplineSurf_Stretch = m_BSplineSurf_Stretch_Array.GetAt(i);
            //首先判断是否在现有的曲线段上
if(CheckIsPicked(point,BSplineSurf_Stretch.BSplineCurve0,m_Picker) == true){
                m_Picker.picktype = pick_bsurf;
                m_Picker.m_BSplineSurf_Stretch = BSplineSurf_Stretch;
                return true;
            }
if(CheckIsPicked(point,BSplineSurf_Stretch.BSplineCurve1,m_Picker) == true){
                m_Picker.picktype = pick_bsurf;
                m_Picker.m_BSplineSurf_Stretch = BSplineSurf_Stretch;
                return true;
            }
            //是否在 BSplineSurf_Stretch.BSplineCurve0 和 SplineCurvel 中间画的两条 B 样条线上
            CArray < CVector,CVector > m_Array_Vector1,m_Array_Vector2;
for(k = 0;k < BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetSize();k++){
                Vt0 = BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(k);
                Vt1 = BSplineSurf_Stretch.BSplineCurve1.m_Array_Vector.GetAt(k);
                Vt2.v_x = Vt0.v_x + (Vt1.v_x - Vt0.v_x)/3.0;
                Vt2.v_y = Vt0.v_y + (Vt1.v_y - Vt0.v_y)/3.0;
                Vt2.v_z = Vt0.v_z + (Vt1.v_z - Vt0.v_z)/3.0;
                Vt3.v_x = Vt0.v_x + (Vt1.v_x - Vt0.v_x) * 2/3.0;
                Vt3.v_y = Vt0.v_y + (Vt1.v_y - Vt0.v_y) * 2/3.0;
                Vt3.v_z = Vt0.v_z + (Vt1.v_z - Vt0.v_z) * 2/3.0;
                m_Array_Vector1.Add(Vt2);
                m_Array_Vector2.Add(Vt3);
            }
            CBSplineCurve BSplineCurve;
            BSplineCurve.m_Array_Vector.Append(m_Array_Vector1);
            if(CheckIsPicked(point,BSplineCurve,m_Picker) == true){
                m_Picker.picktype = pick_bsurf;
                m_Picker.m_BSplineSurf_Stretch = BSplineSurf_Stretch;
                return true;
            }
            BSplineCurve.m_Array_Vector.RemoveAll();
            BSplineCurve.m_Array_Vector.Append(m_Array_Vector2);
            if(CheckIsPicked(point,BSplineCurve,m_Picker) == true){
                m_Picker.picktype = pick_bsurf;
                m_Picker.m_BSplineSurf_Stretch = BSplineSurf_Stretch;
                return true;
            }
            //是否在拉伸的连线直线上
for(k = 0;k < BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetSize() - 3;k++){
                for(j = 0;j < 4;j++){
```

```
if(j == 3&&i!= BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetSize() - 4)
                        continue;
                    t = j * 1.0/3.0;
Vt0 = GetBSplineCurvePt(t, BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(k),
BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(k + 1), BSplineSurf_Stretch.
BSplineCurve0.m_Array_Vector.GetAt(k + 2), BSplineSurf_Stretch.BSplineCurve0.m_Array_
Vector.GetAt(k + 3));
Vt1 = GetBSplineCurvePt(t, BSplineSurf_Stretch.BSplineCurve1.m_Array_Vector.GetAt(k),
BSplineSurf_Stretch.BSplineCurve1.m_Array_Vector.GetAt(k + 1), BSplineSurf_Stretch.
BSplineCurve1.m_Array_Vector.GetAt(k + 2), BSplineSurf_Stretch.BSplineCurve1.m_Array_
Vector.GetAt(k + 3));
                    line.pt1 = CPoint((int)(Vt0.v_x + 0.5),(int)(Vt0.v_y + 0.5));
                    line.pt2 = CPoint((int)(Vt1.v_x + 0.5),(int)(Vt1.v_y + 0.5));
                    if(CheckIsPicked(point,line) == true){
                        m_Picker.picktype = pick_bsurf;
                        m_Picker.m_BSplineSurf_Stretch = BSplineSurf_Stretch;
                        return true;
                    }
                }
            }
        }
        return false;
    }
    else
        return false;
}
```

确定拾取曲面后，在 CopyPicker()函数中将拾取的曲面复制给进行图形操作的拾取变量 m_Picker，代码如下：

```
void CopyPicker(CPicker &m_Picker,CPicker &m_Picker0){
    ...//(原有代码此处省略,下面的代码加在实体复制代码后面)
    else if(m_Picker.picktype == pick_bsurf){ //8.判断是否样条曲面,复制曲面
        m_Picker.m_BSplineSurf_Stretch = m_Picker0.m_BSplineSurf_Stretch;
    }
}
```

同理，绘制拾取图形的函数 DrawPicker 中也需加入绘制曲面的代码：

```
void DrawPicker(CDC * pDC,CPicker &m_Picker,COLORREF crColor,double m_Matrix_V[][4],CBody_
Stretch Body[],int bodyNum = 0,int lineWidth = 0,bool hideFlag = false){
...//(原有代码此处省略,下面的代码加在绘制拾取的实体后面)
else if(m_Picker.picktype == pick_bsurf)//9.绘制拾取的 B 样条曲面
    DrawBSplineSurf(pDC,m_Picker.m_BSplineSurf_Stretch,crColor,lineWidth);
}
```

根据上述代码拾取曲面的效果如图 8.3-11 所示。

对于拾取的曲面可以进行各种操作，例如图形变换，和曲线的图形变换类似，曲面的图形变换通过曲面的控制网格点变换即可实现。由于曲面也是三维图形，因此，曲面变换和三位图形变换相同，例如旋转变换，也可以绕 x、y、z 三个轴旋转。代码如下所示：

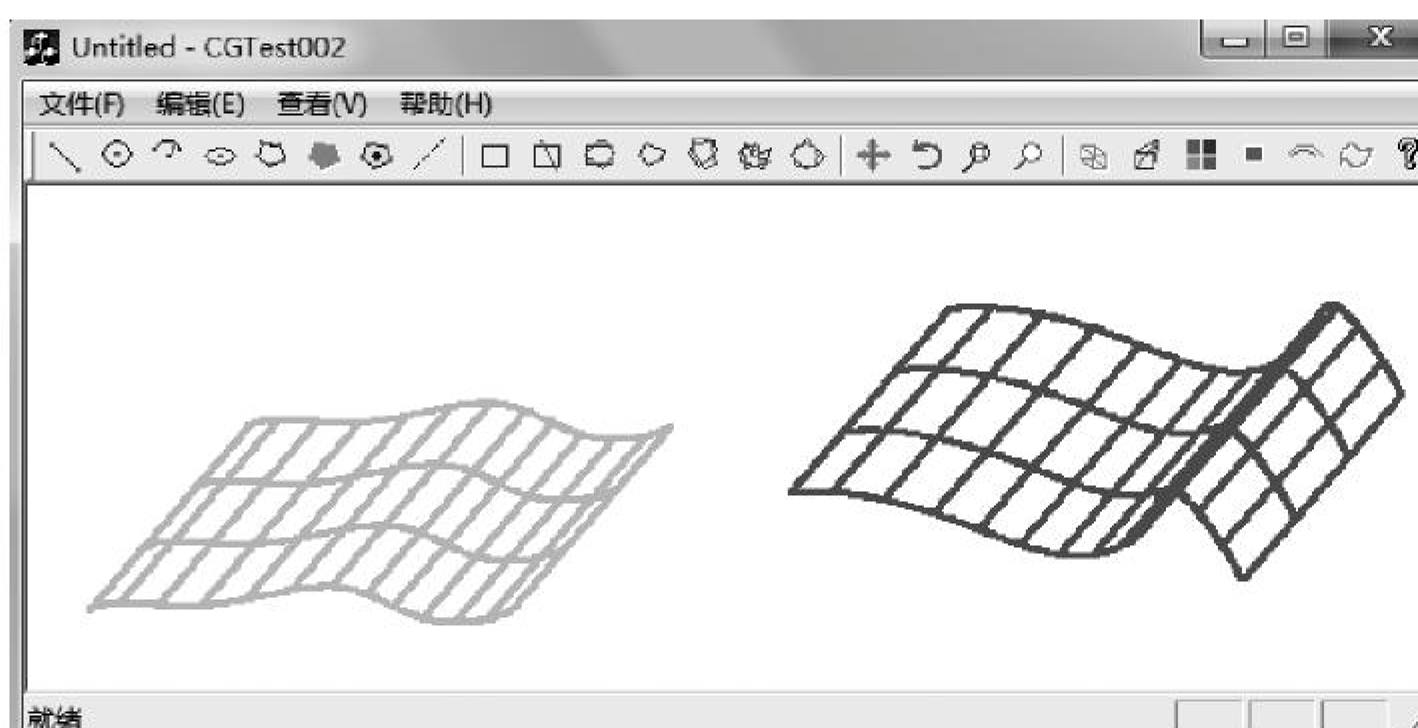


图 8.3-11　曲面拾取

```
void CCGTest002View::Rotate3D(int &m_iAxis,double &angle){
    if(m_Picker.picktype == pick_body){
        ...//对三维图形拉伸体的旋转变换,代码前文已列出,本处省略
    }
    else if(m_Picker.picktype == pick_bsurf){     //曲面旋转变换
//首先获得在曲面集合中的初始曲面
        int i;
        for(i = 0;i < m_BSplineSurf_Stretch_Array.GetSize();i++){
if(m_Picker.m_BSplineSurf_Stretch == m_BSplineSurf_Stretch_Array.GetAt(i)){
m_Picker.m_BSplineSurf_Stretch = m_BSplineSurf_Stretch_Array.GetAt(i);
                break;
            }
        }
        //为了显示,将曲面绕第一个控制顶点旋转
        double m_Matrix[4][4],m_Matrix0[4][4];
        CVector Vt;
        //首先移动到原点
        CPoint3D pt3D;
        Vt = m_Picker.m_BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(0);
        pt3D.x = Vt.v_x;     pt3D.y = Vt.v_y;     pt3D.z = Vt.v_z;
        GetMatrix(m_Matrix,0,pt3D.x * ( - 1),pt3D.y * ( - 1),pt3D.z * ( - 1),0,1);
        GetMatrix(m_Matrix0,m_iAxis,0,0,0,angle,1);          //沿轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);                   //矩阵级联
        GetMatrix(m_Matrix0,0,pt3D.x,pt3D.y,pt3D.z,0,1);  //移回原位置
        MatrixXMatrix(m_Matrix,m_Matrix0);                   //矩阵级联
        GetNewPoint(m_Picker.m_BSplineSurf_Stretch,m_Matrix);
    }
    Invalidate();
}
```

8.4　NURBS 方法

B 样条方法在表示与设计自由型曲线、曲面形状时显示了强大的威力,然而在表示与设计初等曲线、曲面时却遇到了麻烦。因为 B 样条曲线(包括其特例的 Bézier 曲线)不能精确

表示出抛物线外的二次曲线，B样条曲面(包括其特例的Bézier曲面)不能精确表示出抛物面外的二次曲面，而只能给出近似的表示。当把B样条写成有理样条函数的形式时，上述问题就得到了解决。

有理函数是两个多项式之比，有理样条(rational spline)是两个样条函数之比，有理B样条用向量方程表达为

$$P(t)=\frac{\sum_{k=0}^{n} w_k p_k N_{k,m}(t)}{\sum_{k=0}^{n} w_k N_{k,m}(t)}$$

其中 p_k 是控制点位置，w_k 是控制点 p_k 的权因子，其值越大，曲线越靠近控制点 p_k。因此，这时曲线曲面的形状由控制点和权因子共同决定。

在形状描述实践中，有理样条经常以非均匀类型出现，而均匀、准均匀、分段Bézier三种类型可看成是非均匀类型的特例，所以人们习惯称之为非均匀有理B样条(nonuniform rational B-spline)方法，简称NURBS方法。

NURBS曲线与B样条曲线也具有类似的几何性质。

(1) 局部性质。k 阶NURBS曲线上参数为 $t\in[t_i,t_{i+1}]\subset[t_{k-1},t_{n+1}]$ 的一点 $P(t)$ 至多与 k 个控制顶点 p_i 及权因子 $w_j(j=i-k+1,\cdots,i)$ 有关，与其他顶点和权因子无关；另一方面，若移动 k 次NURBS曲线上的一个控制顶点 P_i 或改变所联系的权因子，仅仅影响定义在区间 $[t_i,t_{i+1}]\subset[t_{k-1},t_{n+1}]$ 上那部分曲线的形状。

(2) 凸包性。定义在非零节点区间 $t\in[t_i,t_{i+1}]\subset[t_{k-1},t_{n+1}]$ 上的曲线段位于定义它的 $k+1$ 个控制顶点 $p_{i-k+1},\cdots,p_i$ 的凸包内。整条NURBS曲线位于所有定义各曲线段的控制顶点的凸包的并集内。所有权因子的非负性，保证了凸包性质的成立。

(3) 在仿射与透射变换下的不变性。

(4) 在曲线定义域内有与有理基函数同样的可微性。

用NURBS曲线表示二次曲线，通过调整权因子 w_i 的值即可实现。

假定用定义在三个控制顶点和开放均匀的节点矢量上的二次(三阶)B样条函数来拟合二次曲线，于是，$\boldsymbol{T}=(0,0,0,1,1,1)$，取权因子为

$$w_0=w_2=1$$

$$w_1=\frac{r}{1-r},\quad 0\leqslant r<1$$

则有理B样条的表达式为

$$P(t)=\frac{p_0N_{0,3(t)}+\frac{r}{1-r}p_1N_{1,3(t)}+p_2N_{2,3(t)}}{N_{0,3(t)}+\frac{r}{1-r}N_{1,3(t)}+N_{2,3(t)}}$$

取不同的 r 值得到各种二次曲线如图8.4-1所示：

当 $r>\frac{1}{2}$，$w_1>1$ 时，得到双曲段；

当 $r=\frac{1}{2}$，$w_1>1$ 时，得到抛物线段；

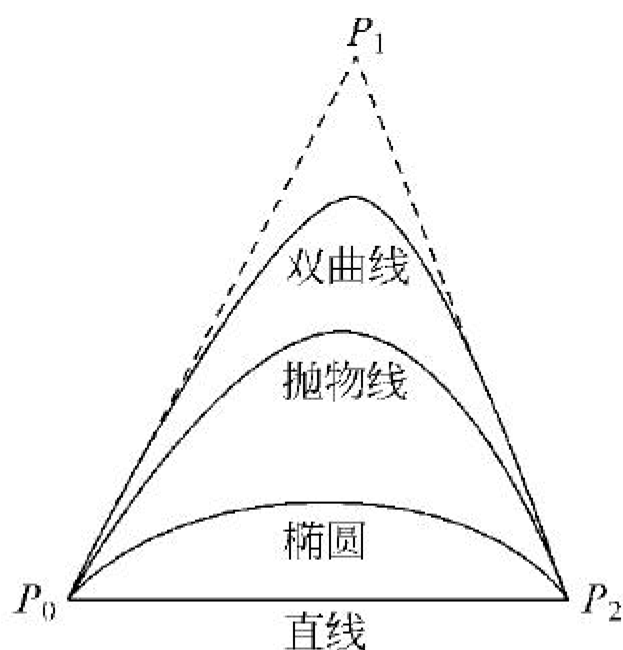


图8.4-1 二次曲线

当 $r<\frac{1}{2}$，$w_1<1$ 时，得到椭圆弧；

当 $r=0$，$w_1=0$ 时，得到直线段。

当选控制点为 $p_0=(0,1)$，$p_1=(1,1)$，$p_2=(1,0)$，$w_1=\cos\alpha$ 时，上式可产生第一象限的1/4单位圆弧，如图8.4-2所示，若要产生圆弧的其他部分，只需改变控制点的位置即可。

同理，NURBS曲面也可以表示各种二次解析曲面，这里对此不再赘述。

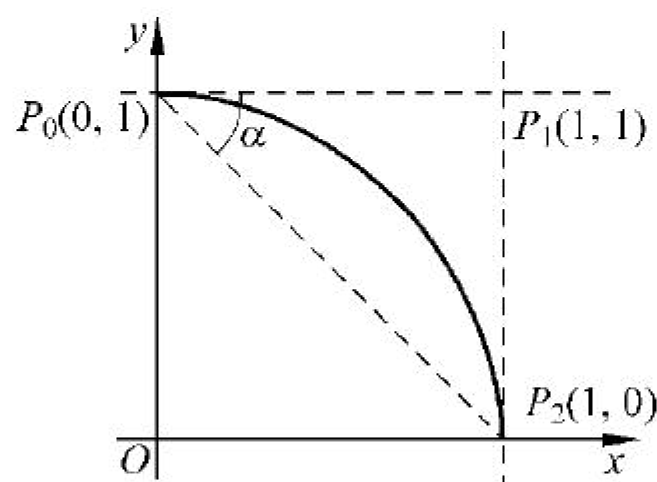


图8.4-2 NURBS表示圆弧

NURBS方法的主要优点如下。

(1) 数学模型统一。该方法既能描述自由曲线和曲面，又能够精确表示圆锥曲线和曲面；非有理B样条、有理与非有理Bézier方法是其特例。因此，它为曲线曲面的CAD/CAM系统提供了一个统一的数学模型和框架。

(2) 形状控制灵活。通过修改控制顶点、权因子或者节点的值，来修改和控制曲线或曲面的形状，为各种形状设计提供了充分的灵活性。

(3) 造型能力强大。具有一系列强有力的几何造型的配套技术(包括节点插入、细分、升阶等)。

(4) 具有仿射变换不变性。对几何变换和投影变换具有不变性。

不过，目前在应用NURBS方法时，还有一些难以解决的问题，具体如下：

(1) 它比传统的曲线曲面定义方法需要更多的存储空间，如空间圆需7个参数(圆心、半径、法矢)，而NURBS定义空间圆需38个参数；

(2) 权因子选择不当会引起畸变；

(3) 对搭接、重叠形状的处理很麻烦；

(4) 反求曲线、曲面上点参数值的算法不稳定。

NURBS是一种非常优秀的建模方式，在高级三维软件中都支持这种建模方式。NURBS可以比传统的网格建模方式更好地控制物体表面的曲线度，从而能够创建出更逼真、生动的造型。NURBS曲线和NURBS曲面在传统的制图领域是不存在的，是为使用计算机进行3D建模而专门建立的。NURBS也是专门做曲面物体的造型方法。可以用它做出各种复杂的曲面造型和表现特殊的效果，如人的皮肤、面貌或流线型的跑车等。关于NURBS的详细理论研究和方法请参阅相关书籍，本书不再赘述。

第 9 章 计算机动画与仿真

计算机图形学的一个重要应用方向是计算机动画与仿真,使用动画与仿真可以清楚地展现一个活灵活现的画面,或者表现一个事件的过程。由于其具有生动的感官效果以及对真实环境的客观模拟,因此,广泛应用在影视广告、游戏娱乐、科研与工程、教育以及军事等众多领域。为了实现计算机动画的逼真性,需要综合使用多种学科和技术。计算机动画与仿真技术以计算机图形学为基础,特别是以图形造型和真实感显示技术(消隐、光照模型、表面纹理)为基础,并涉及图像处理、运动控制原理、视频技术、艺术甚至视觉心理学、生物学、机器人学以及人工智能等领域,计算机动画与仿真因其自身的特点而逐渐成为一门独立的学科。本章主要讲述计算机动画与仿真的相关概念和基本原理,并进行简单的程序实现。

9.1 计算机动画与仿真的概念及基本原理

计算机动画(computer animation)是指采用图形与图像技术,借助计算机生成一系列可供动态实时演播的连续图像的技术。所谓动画就是使一幅图像“活”起来的过程,从而清楚地展现一个活灵活现的画面或者表现一个事件的过程。

计算机动画的基本原理是利用人眼的视觉暂留,把一系列连续的图片快速顺序播放,这时,看起来就是连续动起来的画面。就像早期电影院放映电影的胶片,胶片上静态的人物影像通过连续播放后,人们看到的电影是动态的人物场景。

计算机动画中连续播放的每一幅图像称为帧,计算机动画的本质就是生成一个个帧,并连续显示,其中当前帧是前一帧的部分修改。根据人的视觉滞留特性,为了产生连续运动的感觉,需要连续播放画面。实验证明,动画和电影的画面刷新率为 24 帧/s 时,即每秒放映 24 幅画面,人眼看到的则是连续的画面效果。例如,一分钟长的动画就需要绘制 1440 张连续的帧来表现流畅的画面。目前,计算机动画制作软件很多,不同的动画效果,取决于不同的计算机动画软、硬件的功能。虽然制作的复杂程度不同,但是其采用的基本原理是一致的。

计算机仿真是应用计算机对动画对象的结构、功能和行为以及参与系统控制的人的思维过程和行为进行动态性比较逼真的模仿。计算机仿真更追求对客观世界模拟的真实性,但是它仍然属于计算机动画的应用范畴。

根据表现形式,动画分二维动画和三维动画两种类型。二维动画是平面上的画面,在二维空间上模拟三维空间的效果;三维动画则本身就是三维立体图形,从不同角度得到不同

的形状,然后投影到二维平面上显示。传统的动画制作的技术基础是采用"分层"技术,动画师将运动的物体和静止的背景分别绘制在不同的透明胶片上,然后叠加在一起拍摄,因此传统动画主要实现的是二维动画创作。计算机动画从技术上可以分为计算机辅助动画和造型动画两种。计算机辅助动画的主要用途是辅助动画师制作传统动画;而造型动画则属于三维动画类型,主要是采用计算机图形学以及相关技术,利用计算机强大的运算能力来模拟现实,这个过程需要完成动画对象的建模、动画动作设计以及场景渲染等步骤。建模是以点、线、面的方式建立动画对象的几何信息,动作设计是按照运动规律和动力学模拟等方式设计动画对象的运动轨迹路线,场景渲染则通过纹理映射、光照模型等方式显示场景。

计算机动画是计算机图形学的一个重要应用方向,也是图形学的一个研究重点,以计算机动画为基础的虚拟现实(VR)技术是现在的一个研究热点。计算机动画主要研究与动画有关的造型、绘制、合成以及运动控制等技术,尽管实体造型和自由曲线曲面造型技术在CAD和CAGD中得到了广泛的研究,但计算机动画对传统的实体、曲面造型提出了一些新的要求。一方面,计算机动画中场景造型的精度不必像工业设计那样高;另一方面,对造型工具的灵活性及景物运动的可控性提出了更高的要求。这导致出现许多针对动画应用而设计的造型技术,如隐函数曲面造型技术以及离散曲面造型技术等。除此之外,由于其简单性和兼容性,多边形网格模型在计算机动画系统中得到了广泛的重视。绘制本身是真实感图形的主要研究内容,但随着动画技术的发展,对传统的真实感图形绘制技术必须予以改造,使之满足动画的需要。

9.2 计算机动画与仿真的实现方法

根据帧生成的方法不同,计算机动画可以有两种实现方法:逐帧动画和实时动画。

9.2.1 逐帧动画

逐帧动画(frame by frame)是一种常见的动画形式,即在时间轴上逐帧绘制帧内容,由于是一帧一帧地画,所以逐帧动画具有非常大的灵活性,几乎可以表现任何想表现的内容。其原理是在"连续的关键帧"中分解动画动作,也就是在时间轴的每帧上逐帧绘制不同的内容,使其连续播放而成动画。因为逐帧动画的帧序列内容不一样,所以,给动画制作增加了负担,而且最终输出的文件量也很大;其优点是,它类似于电影的播放模式,很适合表现细腻的动画。例如:人物或动物急剧转身,头发及衣服的飘动,走路、说话以及精致的3D效果等。

创建逐帧动画有以下几种方法。

(1) 用导入的静态图片建立逐帧动画。用jpg、png等格式的静态图片连续导入Flash中,就会建立一段逐帧动画。

(2) 绘制矢量逐帧动画。用鼠标或压感笔在场景中一帧帧地画出帧内容。

(3) 文字逐帧动画。用文字作帧中的元件,实现文字跳跃、旋转等特效。

(4) 导入序列图像。可以导入gif序列图像、swf动画文件或者利用第三方软件(如

swish、swift 3D 等)产生的动画序列。

在动画中,重要的帧称为关键帧,关键帧之间的帧称为中间帧。逐帧动画中的中间帧由计算机完成,计算机使用关键帧的算法自动生成中间各帧。最常用的关键帧算法是插值算法。所有影响画面图像的参数,例如位置、旋转角度、纹理等都是关键帧参数。

中间帧生成的插值算法的一般步骤如下。

(1) 分解。首先将关键帧画面分解成几部分,每一部分包含一个关键图形,这些关键图形将作为生成操作的处理单元。然后,将每一部分的关键图形又分解成若干个笔画,这些笔画是可见线段的组合序列。

(2) 预处理。如果两关键图形的笔画(或折线、曲线)数量不相等,则进行预处理,其目的是使两画面的笔画数量相等。所有的对应笔画均需要有相同数量的点。

(3) 插值。要计算两关键帧图形之间的中间图形,需在两对应画面图形之间进行插值计算。插值一般有两种方式: 线性插值和非线性插值。线性插值可实现平稳的过渡效果,非线性插值则可以实现某种特殊的加速度效果。在插值计算过程中,可以采用四个不同的法则,它们可使物体在中间的画面中分别以等速、加速、减速或先加速后减速的方式运动。

线性插值算法计算方法简单、速度快,但存在一些问题: 第一,图形变换中每个点都是沿着直线运动,而且每个点的运动规则都相同,生成的动画显得生硬、不自然; 第二,运动设计中要求几个关键帧画面,这些画面造成了运动的不连续; 第三,若关键帧画面之间有旋转、扭曲等分量,线性插值就会产生失真。

骨架法是改善线性插值的一种方法。该方法将图形抽象简化为由简单的骨架构成,而不是用图形本身来作为插值的依据。骨架,或称线条图,是由几个点组成的图形或人物形态的简单描述,它描述要求的运动形式。使用这种方法,动画创作人员只需控制一些由骨架生成的关键图,然后计算机再由这些关键骨架图生成中间骨架图。由于骨架图较简单,也相似,因此能得到较好的中间图。

采用线性插值法计算物体形状的变化是很方便的,当给出两幅关键图形后,就能生成效果很好的中间图画。但是,由于前面谈到的线性插值的几个问题,它不大适合于物体位置改变的运动场合。这个问题可应用运动物体所遵循的物理定律来解决,将路径描述与插值算法相结合也能较方便地解决。动态图可以依靠连续变动静态图来得到。

9.2.2 实时动画

实时动画(real-time)也称算法动画,它是采用各种算法来实现运动物体的运动控制,在实时动画中,计算机一边计算一边显示来产生动画效果。实时动画一般不包含大量的动画数据,而是对有限的数据进行快速处理,并将结果随时显示出来。实时动画的响应时间与许多因素有关,如计算机的运算速度、软硬件处理能力、景物的复杂程度、画面的大小等。

在实时动画中,一种最简单的运动形式是对象的移动,它是指屏幕上一个局部图像或对象在二维平面上沿着某一固定轨迹运动。

实时动画是最灵活的动画,但也是最慢的。由于在进行动画展示的同时绘制每一幅图像,因此可以根据需要动态地改变下一幅图像内容,也就是最具有交互性。实时动画的交互特性使之成为三维环境中移动模拟的有利候选者。

实时动画是采用算法实现对物体的运动控制或模拟摄像机的运动控制，一般适合于三维情形。根据不同的算法可分为以下几种。

（1）运动学算法。由运动学方程确定物体的运动轨迹和速率。

（2）动力学算法。从运动的动因出发，由力学方程确定运动形式。

（3）反向运动学算法。已知链接物末端的位置和状态，反求运动方程以确定运动形式。

（4）反向动力学算法。已知链接物末端的位置和状态，反求动力学方程以确定运动形式。

（5）随机运动算法。在某些场合下加进运动控制的随机因素。

尽管实时动画达不到帧动画那样的速度，但良好的交互性使得它成为动画设计的最佳选择。在动画展示的同时生成每一幅图像，可以很容易地使用键盘来改变进行动画展示的模型的形状、位置和其他特征。改善实时动画效果有以下方法。

（1）越简单越好。尽量使图像简单。计算机生成图像的时间越长，动画的速度就越慢。在实时动画中，模型的动作决定了图形的可信程度，也就是说，绘制模型的细节并不太重要。重要的是动作。如果我们注重图形的细节，那么最好使用帧动画或BITBLIT动画。

（2）使用背景缓冲区。如果需要复杂的背景，比如远处的群山或城堡，可以使用另一个隐藏页，不妨称之为背景缓冲区，把背景画好后存储在该页中。动画序列的每一帧图像用下面介绍的方法产生：先把背景缓冲区的内容复制到作图页中；然后在作图页上绘制图像；最后切换显示页和作图页，这样含有背景的图像就被显示到屏幕上。

（3）优化运动轨迹函数。尽量减少复杂公式的计算时间，例如正余弦等费时的计算尽量调整为增量运算，把一些复杂的公式计算分解为减少计算量的多部分组合来实现。

（4）使用线性模型。尽量采用线性模型，复杂的曲面造型计算十分费时，平面线性造型相对计算量较少，因此，在不影响效果的前提下，应尽量选择线性造型模型来表达。

9.3　计算机实时动画实践

由于计算机实时动画方法并不需要事先存储动画帧，而是在动画展示过程中实时计算获得每一个画面，那么，对于本书前文利用图形学原理生成的各种图形，如三维拉伸体、曲面等，如果将其作为动画展示对象，在一定的时间内连续进行图形变换操作，如平移变换或者旋转变换，并连续显示每一次图形变换的结果，这样就可以产生实时动画的效果。

为了实现上述的图形实时动画方法，可以利用前面章节已经完成的应用程序CGTest002，首先在程序中创建一个如图9.3-1所示的CAnimationDlg非模式对话框作为主要设置窗口。在该对话框内设置动画对象、动画运动轨迹，以及设置开始和停止动画的播放。

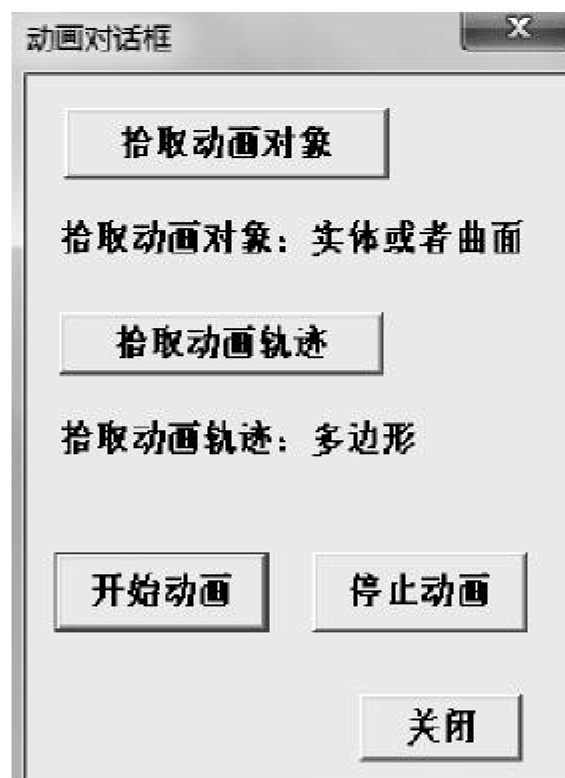


图9.3-1　动画设置对话框

首先，在CAnimationDlg对话框的类结构中，增加动画对象和动画轨迹两个变量，动画对象和动画轨迹可以通过在屏幕上拾取图形获得，因此，这两个变量都定义为CPicker拾取类的

类型。

为了实现交互操作，在对话框中设置两个按钮 IDC_BUTTON_PICK_OBJECT 和 IDC_BUTTON_PICK_TRAIL 来拾取动画对象和拾取动画轨迹，并设置两个静态文本控件 IDC_STATIC_OBJEC 和 IDC_STATIC_TRAIL 说明拾取状态。

在动画展示时，为了实现连续的图形变换操作，需要在应用程序中设置一个计时器。开始动画展示时，计时器开始计时。设置计时的时间间隔，例如 100ms 一个间隔，在每一次计时的间隔，动画对象图形变换到一个新位置并实时显示，这样，连续播放图形变换的结果就是实时动画的效果。在 VC 程序中，计时器对应的消息是 WM_TIMER，在“建立类向导”里映射 WM_TIMER 对应的消息函数 OnTimer()。当在程序中调用 SetTimer()函数时，启动计时器，SetTimer()函数的第一个参数对应一个计时器序号，第二个参数是设置计时器的时间间隔，例如 SetTimer(1,100,NULL)就设置并启动了一个 100ms 间隔的计时器。计时器启动后，在每一个计时间隔都会调用消息函数 OnTimer()，将动画对象的每一次图形变换代码放在 OnTimer()中，并实时显示，即可实现动画展示。当停止动画展示时，调用 KillTimer(1)函数，参数值 1 为对应的计时器序号，则计时器停止计时，程序不再调用 OnTimer()函数，动画展示停止。

本应用程序实现动画展示的思路是，首先在显示屏幕上拾取动画对象和动画轨迹，为了减少程序处理的代码量，动画对象只拾取拉伸实体和 B 样条曲面，动画轨迹只拾取多边形作为运动轨迹。动画展示开始时，动画对象首先平移到动画轨迹的起点，然后再沿动画轨迹的路线平移，动画对象在平移的同时绕自身旋转，自身旋转点取其边界包围盒的中心点。

在应用程序的 CCGTest002View 类中建立动画对话框 CAnimationDlg 的指针变量，注意：在 CCGTest002View 视图的构造函数中创建该指针对象，在析构函数中释放该指针对象。在应用程序的工具栏中建立一个用于打开动画对话框的工具条。动画对话框打开的代码如下：

```
void CCGTest002View::OnAnimation() {                    //动画
    if(this->m_AnimationDlg->GetSafeHwnd() == NULL)
        this->m_AnimationDlg->Create();
    else
        this->m_AnimationDlg->ShowWindow(TRUE);
    this->m_iFlag = 18;                                 //动画对应的标识为 18
}
```

动画设置对话框 CAnimationDlg 的头文件 AnimationDlg.h 代码如下，其中粗体代码部分为手动增加的变量等代码，其他相关代码可以通过类向导自动生成：

```
#if !defined(AFX_ANIMATIONDLG_H__A8B7C766_2285_4ACE_9F1F_D73FC47B5FBE__INCLUDED_)
#define AFX_ANIMATIONDLG_H__A8B7C766_2285_4ACE_9F1F_D73FC47B5FBE__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif
#include "BasicClass.h"
class CCGTest002View;
class CAnimationDlg : public CDialog{
//Construction
```

```
public:
    CAnimationDlg(CWnd* pParent = NULL);          //standard constructor
    BOOL Create();
    CCGTest002View *m_pView;
    int m_Flag;                                   //动画标识符
    int m_iAnimationFlag;                         //动画计数
    CPicker m_AnimationObject;                    //动画对象,只允许是拉伸体和曲面
    CPicker  m_AnimationTrail;                    //动画轨迹,只允许是多边形
    CPoint3D Pt_Object_0,Pt_Object_1,Pt_Object;   //动画对象起始、终止及现在位置
    int m_DisNum;                                 //记录每个轨迹段的细分数量
    int m_iTrailNo;                               //轨迹段
//Dialog Data
    //{{AFX_DATA(CAnimationDlg)
    enum { IDD = IDD_DIALOG_ANIMATION };
    CString  m_strObject;
    CString  m_strTrail;
    //}}AFX_DATA
//Overrides
    //ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAnimationDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);     //DDX/DDV support
    //}}AFX_VIRTUAL
//Implementation
protected:
    //Generated message map functions
    //{{AFX_MSG(CAnimationDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnStop();
    afx_msg void OnStart();
    afx_msg void OnClose();
    afx_msg void OnButtonPickObject();
    afx_msg void OnButtonPickTrail();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
#endif
```

CAnimationDlg 的执行文件 AnimationDlg.cpp 代码如下:

```
#include "stdafx.h"
#include "CGTest002.h"
#include "AnimationDlg.h"
#include "math.h"
#include "CGTest002View.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
///////////////////////////////////////////////////////////////////////////////
//CAnimationDlg dialog
CAnimationDlg::CAnimationDlg(CWnd * pParent /* = NULL */)
    : CDialog(CAnimationDlg::IDD, pParent){
    //{{AFX_DATA_INIT(CAnimationDlg)
    m_strObject = _T("");
    m_strTrail = _T("");
    //}}AFX_DATA_INIT
    m_Flag = 0;
    m_iAnimationFlag = 0;
}
BOOL CAnimationDlg::Create(){
    return CDialog::Create(CAnimationDlg::IDD);
}
void CAnimationDlg::DoDataExchange(CDataExchange * pDX){
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAnimationDlg)
    DDX_Text(pDX, IDC_STATIC_OBJECT, m_strObject);
    DDX_Text(pDX, IDC_STATIC_TRAIL, m_strTrail);
    //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CAnimationDlg, CDialog)
    //{{AFX_MSG_MAP(CAnimationDlg)
    ON_WM_TIMER()
    ON_BN_CLICKED(IDC_STOP, OnStop)
    ON_BN_CLICKED(ID_START, OnStart)
    ON_WM_CLOSE()
    ON_BN_CLICKED(IDC_BUTTON_PICK_OBJECT, OnButtonPickObject)
    ON_BN_CLICKED(IDC_BUTTON_PICK_TRAIL, OnButtonPickTrail)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
///////////////////////////////////////////////////////////////////////////////
//CAnimationDlg message handlers
BOOL CAnimationDlg::OnInitDialog() {
    CDialog::OnInitDialog();
    if(this->m_AnimationObject.picktype!=pick_body||this->m_AnimationObject.picktype!=
pick_bsurf)
        this->m_strObject="拾取动画对象：实体或者曲面";
    if(this->m_AnimationTrail.picktype!=pick_polyline)
        this->m_strTrail="拾取动画轨迹：多边形";
    UpdateData(FALSE);
    return TRUE;      //return TRUE unless you set the focus to a control
                      //EXCEPTION: OCX Property Pages should return FALSE
}
void CAnimationDlg::OnTimer(UINT nIDEvent)
{//计算移动后的中心位置,并进行图形变换
    m_iAnimationFlag++;
Pt_Object.x=Pt_Object_0.x+(Pt_Object_1.x-Pt_Object_0.x)*m_iAnimationFlag/m_DisNum;
Pt_Object.y=Pt_Object_0.y+(Pt_Object_1.y-Pt_Object_0.y)*m_iAnimationFlag/m_DisNum;
Pt_Object.z=Pt_Object_0.z+(Pt_Object_1.z-Pt_Object_0.z)*m_iAnimationFlag/m_DisNum;
this->m_pView->Animation(m_AnimationObject,Pt_Object,(Pt_Object_1.x-Pt_Object_0.x)/m_
```

```
DisNum,(Pt_Object_1.y - Pt_Object_0.y)/m_DisNum,(Pt_Object_1.z - Pt_Object_0.z)/m_DisNum,
1.0,1.0,1.0);                                           //图形变换
    if(m_iAnimationFlag == m_DisNum){                   //计算下一轨迹段
        Pt_Object_0 = Pt_Object;
if(m_iTrailNo < m_AnimationTrail.m_PolyLine.m_PolyLine_array_Out.GetSize() - 1)
            m_iTrailNo++;
        else
            m_iTrailNo = 0;                             //全部动画完后,再从头循环动画
Pt_Object_1.x = this -> m_AnimationTrail.m_PolyLine.m_PolyLine_array_Out.GetAt(m_iTrailNo).
pt2.x;
Pt_Object_1.y = this -> m_AnimationTrail.m_PolyLine.m_PolyLine_array_Out.GetAt(m_iTrailNo).
pt2.y;
        Pt_Object_1.z = 0;
        //设置距离3个像素为一个间隔
m_DisNum = sqrt((Pt_Object_1.x - Pt_Object_0.x) * (Pt_Object_1.x - Pt_Object_0.x) + (Pt_Object_
1.y - Pt_Object_0.y) * (Pt_Object_1.y - Pt_Object_0.y))/3.0;
        if(m_DisNum == 0) m_DisNum = 1;
        m_iAnimationFlag = 0;
    }
    this -> m_pView -> Invalidate();
    CDialog::OnTimer(nIDEvent);
}
void CAnimationDlg::OnStop() {                          //停止动画
    if(m_Flag == 1){
        KillTimer(1);                                   //关闭计时器,停止动画
        m_Flag = 0;
        m_iAnimationFlag = 0;
    }
}
//计算包围盒的两个边界极值点
void GetMinMaxPt(CPoint3D &Pt,CPoint3D &Ptmin,CPoint3D &Ptmax){
    if(Ptmin.x > Pt.x) Ptmin.x = Pt.x;
    if(Ptmin.y > Pt.y) Ptmin.y = Pt.y;
    if(Ptmin.z < Pt.z) Ptmin.z = Pt.z;
    if(Ptmax.x < Pt.x) Ptmax.x = Pt.x;
    if(Ptmax.y < Pt.y) Ptmax.y = Pt.y;
    if(Ptmax.z < Pt.z) Ptmax.z = Pt.z;
}
void CAnimationDlg::OnStart()
{//开始动画,首先,在动画对象和轨迹起始点之间作为第一个动画轨迹段
    if(m_Flag == 0)   {
        if(this -> m_AnimationObject.picktype == pick_none) return;
        //计算动画对象的中心点
        CPoint3D Pt0min,Pt0max,Pt1min,Pt1max;
        double dbl;
        if(this -> m_AnimationObject.picktype == pick_body){    //动画对象是拉伸体
            //首先,计算拉伸体所在的空间边界包围盒
            Pt0min = this -> m_AnimationObject.m_Body_Stretch. EdgePlane[0].loop_out.GetAt(0).
Pt1_3D;
            Pt0max = this -> m_AnimationObject.m_Body_Stretch. EdgePlane[1].loop_out.GetAt(0).
Pt1_3D;
```

```
            if(Pt0min.x > Pt0max.x){
                dbl = Pt0min.x;
                Pt0min.x = Pt0max.x;
                Pt0max.x = dbl;
            }
            if(Pt0min.y > Pt0max.y){
                dbl = Pt0min.y;
                Pt0min.y = Pt0max.y;
                Pt0max.y = dbl;
            }
            if(Pt0min.z > Pt0max.z){
                dbl = Pt0min.z;
                Pt0min.z = Pt0max.z;
                Pt0max.z = dbl;
            }
for(int i = 0;i < m_AnimationObject.m_Body_Stretch.EdgePlane[0].loop_out.GetSize();i++){
    GetMinMaxPt(this->m_AnimationObject.m_Body_Stretch.EdgePlane[0].loop_out.GetAt(i).
Pt2_3D,Pt0min,Pt0max);
    GetMinMaxPt(this->m_AnimationObject.m_Body_Stretch.EdgePlane[1].loop_out.GetAt(i).
Pt2_3D,Pt0min,Pt0max);
            }
            Pt_Object_0.x = (int)(0.5 + (Pt0min.x + Pt0max.x)/2.0);
            Pt_Object_0.y = (int)(0.5 + (Pt0min.y + Pt0max.y)/2.0);
            Pt_Object_0.z = (int)(0.5 + (Pt0min.z + Pt0max.z)/2.0);
        }
        else {//动画对象是曲面,首先计算曲面的空间边界包围盒
Pt0min = m_AnimationObject.m_BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(0);
Pt0max = m_AnimationObject.m_BSplineSurf_Stretch.BSplineCurve1.m_Array_Vector.GetAt(0);
            if(Pt0min.x > Pt0max.x){
                dbl = Pt0min.x;
                Pt0min.x = Pt0max.x;
                Pt0max.x = dbl;
            }
            if(Pt0min.y > Pt0max.y){
                dbl = Pt0min.y;
                Pt0min.y = Pt0max.y;
                Pt0max.y = dbl;
            }
            if(Pt0min.z > Pt0max.z){
                dbl = Pt0min.z;
                Pt0min.z = Pt0max.z;
                Pt0max.z = dbl;
            }
            for(int i = 1;i < this->m_AnimationObject.m_BSplineSurf_Stretch.BSplineCurve0.
m_Array_Vector.GetSize();i++) {
    Pt1min = this->m_AnimationObject.m_BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.
GetAt(i);
                GetMinMaxPt(Pt1min,Pt0min,Pt0max);
Pt1min = this->m_AnimationObject.m_BSplineSurf_Stretch.BSplineCurve1.m_Array_Vector.GetAt
(i);
                GetMinMaxPt(Pt1min,Pt0min,Pt0max);
```

```
                }
                Pt_Object_0.x = (int)(0.5 + (Pt0min.x + Pt0max.x)/2.0);
                Pt_Object_0.y = (int)(0.5 + (Pt0min.y + Pt0max.y)/2.0);
                Pt_Object_0.z = (int)(0.5 + (Pt0min.z + Pt0max.z)/2.0);
            }
            //计算拾取轨迹第一个点
Pt_Object_1.x = m_AnimationTrail.m_PolyLine.m_PolyLine_array_Out.GetAt(0).pt1.x;
Pt_Object_1.y = m_AnimationTrail.m_PolyLine.m_PolyLine_array_Out.GetAt(0).pt1.y;
            Pt_Object_1.z = 0;
            //按3个像素距离细分每一个轨迹段
m_DisNum = sqrt((Pt_Object_1.x - Pt_Object_0.x) * (Pt_Object_1.x - Pt_Object_0.x) + (Pt_Object
_1.y - Pt_Object_0.y) * (Pt_Object_1.y - Pt_Object_0.y))/3.0;
            if(m_DisNum == 0) m_DisNum = 1;
            this -> m_pView -> m_Picker.picktype = pick_none;
            m_iTrailNo = - 1;                  //第 - 1 段轨迹
            SetTimer(1,40,NULL);               //设置了一个 40ms 的定时器,启动计时器,开始动画
            m_Flag = 1;
            m_iAnimationFlag = 0;
        }
    }
void CAnimationDlg::OnClose()
{//关闭动画对话框时,停止动画
    if(m_Flag == 1){
        KillTimer(1);                          //关闭计时器,停止动画
        m_Flag = 0;
        m_iAnimationFlag = 0;
        this -> m_pView -> Invalidate();
    }
    CDialog::OnClose();
}
void CAnimationDlg::OnButtonPickObject()
{
    this -> m_pView -> m_iFlag = 19;           //设置在视图 View 里拾取动画对象
}
void CAnimationDlg::OnButtonPickTrail()
{
    this -> m_pView -> m_iFlag = 20;           //设置在视图 View 里拾取动画轨迹
}
```

在动画对话框的计时消息函数 OnTimer()中调用的 CCGTest002View 中的 Animation()图形变换函数代码为:

```
void CCGTest002View::Animation(CPicker& m_Obj,CPoint3D &Pt_Object,double x_dis,double y_dis,
double z_dis,double x_rAngle,double y_rAngle,double z_rAngle){
    double m_Matrix[4][4];
    GetMatrix(m_Matrix,0,x_dis,y_dis,z_dis,0,0);    //iFlag:0 移动
    GetNewAnimate(m_Obj,m_Matrix);                  //生成变换后的点
    int iAxis = 1;                                  //沿 x 轴旋转变换
    AnimationRotate(m_Obj,Pt_Object,iAxis,x_rAngle);//旋转变换
    iAxis = 2;                                      //沿 y 轴旋转变换
    AnimationRotate(m_Obj,Pt_Object,iAxis,y_rAngle);
```

```
    iAxis = 3;                                          //沿 z 轴旋转变换
    AnimationRotate(m_Obj,Pt_Object,iAxis,z_rAngle);
    if(m_Obj.picktype == pick_body){//修改原拉伸体的位置
        for(int i = 0;i < num_Body;i++) {
            if(m_Body[i] == m_Obj.m_Body_Stretch){
                m_Body[i] = m_Obj.m_Body_Stretch;
                break;
            }
        }
    }
    else    {//修改原曲面的位置
        for(int i = 0;i < m_BSplineSurf_Stretch_Array.GetSize();i++) {
            if(m_Obj.m_BSplineSurf_Stretch == m_BSplineSurf_Stretch_Array.GetAt(i)){
                m_BSplineSurf_Stretch_Array.RemoveAt(i);
m_BSplineSurf_Stretch_Array.InsertAt(i,m_Obj.m_BSplineSurf_Stretch);
                break;
            }
        }
    }
    m_Picker.picktype = pick_none;
    Invalidate();
}
```

其中移动变换矩阵后生成新点的函数 GetNewAnimate()代码如下：

```
void CCGTest002View::GetNewAnimate(CPicker& m_Obj,double m_Matrix[][4]){
    if(m_Obj.picktype == pick_bsurf)
        GetNewPoint(m_Obj.m_BSplineSurf_Stretch,m_Matrix);
    else
        GetNewPoint(m_Obj.m_Body_Stretch,m_Matrix);
}
```

旋转变换并生成新点的函数 AnimationRotate()代码如下：

```
void CCGTest002View::AnimationRotate(CPicker& m_Obj, CPoint3D &pt3D, int &m_iAxis, double
&angle){
    if(m_Obj.picktype == pick_body) {
        double m_Matrix[4][4],m_Matrix0[4][4];
        //首先移动到原点
        GetMatrix(m_Matrix,0,pt3D.x * ( - 1),pt3D.y * ( - 1),pt3D.z * ( - 1),0,1);
        GetMatrix(m_Matrix0,m_iAxis,0,0,0,angle,1);          //沿轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);                   //矩阵级联
        GetMatrix(m_Matrix0,0,pt3D.x,pt3D.y,pt3D.z,0,1);     //移回原位置
        MatrixXMatrix(m_Matrix,m_Matrix0);                   //矩阵级联
        //实体乘以变换矩阵,得新点
        GetNewPoint(m_Obj.m_Body_Stretch,m_Matrix);
    }
    else if(m_Obj.picktype == pick_bsurf){
        //为了显示,将实体绕第一个点旋转
        double m_Matrix[4][4],m_Matrix0[4][4];
        CVector Vt;
        //首先移动到原点
```

```
        CPoint3D pt3D;
        Vt = m_Picker.m_BSplineSurf_Stretch.BSplineCurve0.m_Array_Vector.GetAt(0);
        pt3D.x = Vt.v_x;    pt3D.y = Vt.v_y;    pt3D.z = Vt.v_z;
        GetMatrix(m_Matrix,0,pt3D.x * ( - 1),pt3D.y * ( - 1),pt3D.z * ( - 1),0,1);
        GetMatrix(m_Matrix0,m_iAxis,0,0,0,angle,1);          //沿轴旋转
        MatrixXMatrix(m_Matrix,m_Matrix0);                   //矩阵级联
        GetMatrix(m_Matrix0,0,pt3D.x,pt3D.y,pt3D.z,0,1);     //移回原位置
        MatrixXMatrix(m_Matrix,m_Matrix0);                   //矩阵级联
        GetNewPoint(m_Picker.m_BSplineSurf_Stretch,m_Matrix);
    }
    this->m_iFlag = -1;
    Invalidate();
}
```

在动画设置对话框中，单击“拾取动画对象”或者“拾取动画轨迹”按钮时，分别设置CCGTest002View中标识m_iFlag=19或者m_iFlag=20，这样就可以在视图中拾取动画对象和动画轨迹。在OnMouseMove()中增加鼠标移动拾取动画对象或者拾取动画轨迹的代码如下：

```
void CCGTest002View::OnMouseMove(UINT nFlags, CPoint point) {
    ...//(原有代码此处省略)
    else if(this->m_iFlag == 19){//拾取动画对象: 实体或者曲面
        int iflag = 0;
        if(CheckIsPicked(point,m_Body,num_Body,m_Picker0) == true)     //是否在实体上
            iflag = 1;
        else
if(CheckIsPicked(point,m_BSplineSurf_Stretch_Array,m_Picker0) == true)
            iflag = 1;
        else
            iflag = 0;
    }
    else if(this->m_iFlag == 20){     //拾取动画的轨迹,是否在多边形边上
        int iflag = 0;
if(CheckIsPicked(point,m_PolyLine,iPolyLine,this->m_Picker0) == true)
            iflag = 1;
        else
            iflag = 0;
    }
}
```

在OnLButtonDown()中增加确定拾取动画对象和动画轨迹代码如下：

```
void CCGTest002View::OnLButtonDown(UINT nFlags, CPoint point) {
    ...//(原有代码此处省略)
    else if(this->m_iFlag == 19){
        //拾取动画的实体或者曲面
        if(m_Picker0.picktype!= pick_body&&m_Picker0.picktype!= pick_bsurf)
            return;
        CopyPicker(this->m_AnimationDlg->m_AnimationObject,m_Picker0);
        this->m_AnimationDlg->m_strObject = "已拾取";
        this->m_AnimationDlg->UpdateData(FALSE);
```

```
    }
    else if(this->m_iFlag==20){
        //拾取动画的轨迹多边形
        if(m_Picker0.picktype!=pick_polyline)
            return;
        CopyPicker(this->m_AnimationDlg->m_AnimationTrail,m_Picker0);
        this->m_AnimationDlg->m_strTrail="已拾取";
        this->m_AnimationDlg->UpdateData(FALSE);
    }
}
```

图 9.3-2 所示为采用上述程序实现的动画效果图。动画对象为拾取的一个拉伸实体，动画轨迹拾取的是一个多边形，开始动画后，拉伸实体先运动到多边形的起始点，然后，沿着多边形的边界轨迹进行平移运动，在平移运动的同时绕自身旋转。

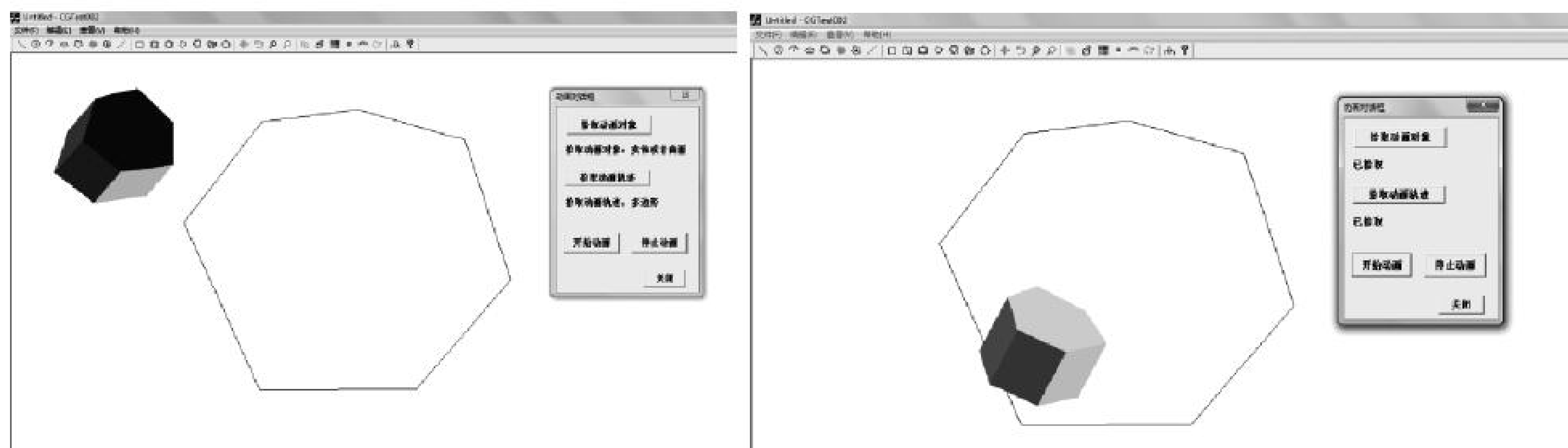


图 9.3-2 图形实时动画

第二部分

第10章 基于OpenGL的图形开发技术

OpenGL是美国SGI公司开发的一套实现计算机图形应用的接口开发包，它将开发计算机图形应用程序所需的底层图形功能——例如基本图形生成、图形变换、消隐、光照、材质、反走样、纹理映射、曲线曲面造型、图像处理以及图形的交互选取等复杂的计算机图形学算法——封装成一系列的图形库函数。这样，在实现具体的图形功能时，可以直接在OpenGL提供的图形库函数基础上进行开发，而不用把大量时间花在基本图形算法的编程上，从而极大提高开发效率。另外，OpenGL支持计算机的各种图形硬件，这样，利用OpenGL作为图形软件接口开发的计算机图形应用程序具有更高的性能和质量。广大计算机厂商也将OpenGL与自己的软硬系统相集成，OpenGL可以兼容多种操作系统平台如Windows、UNIX、Linux以及MacOS等，甚至兼容手机移动设备采用的Android和iOS等系统，所以OpenGL具有开放性、跨平台以及与硬件无关的特点。OpenGL可以实现四个方面的功能：一是几何图形对象的绘制，从二维基本图形到三维形体，并可实现消隐、光照等复杂真实感图形显示及动画等；二是具有一定的图像处理功能；三是实现比较复杂的纹理映射；四是曲线曲面绘制。因此，OpenGL具有强大的图形功能，在计算机图形开发领域得到了广泛的应用。

由于OpenGL是一组应用开发接口（即API），所以，要使用OpenGL开发图形程序，必须首先将开发环境配置为支持OpenGL的状态，在使用OpenGL的图形库函数时，要严格遵守其要求的接口规范。本章将介绍OpenGL在VC6.0环境下的开发设置方法、基本图形的OpenGL编程方法以及实现相关的图形功能，以便掌握基本的OpenGL图形开发方法。

10.1 OpenGL开发环境配置及相关规范介绍

10.1.1 VC6.0环境OpenGL配置方法

在开发OpenGL程序时，由于兼容性，Windows操作系统和VC6.0环境中已经自带了OpenGL的核心库文件，使用时直接加入头文件和编译好的库文件即可，但是还有一些扩展的库文件（如GLUT工具包）在编写应用程序时也会用到，需要下载，并放到系统中以便使用。Windows环境下的GLUT最新版下载地址为（也可通过其他途径下载）：

```
http://www.opengl.org/resources/libraries/glut/glut37.zip
```

将下载的压缩包解开，将得到5个文件。将其中的glut.h文件放到计算机硬盘系统盘符下的“\Program Files (x86)\Microsoft Visual Studio\VC98\Include\GL”文件夹下；将glut.lib和glut32.lib文件放到系统盘符下的“\Program Files (x86)\Microsoft Visual Studio\VC98\Lib”文件夹下；将glut.dll和glut32.dll文件放到系统盘符下的“\Windows\System32”文件夹下（对于64位操作系统，这两个文件放在“\Windows\SysWOW64”文件夹下）。注意：当发布应用程序时，需要将glut.dll和glut32.dll文件放入安装包中，和应用程序的执行文件一起复制到其他计算机上。

VC6.0下有两种OpenGL的使用方式，第一种是创建一个Win32控制台应用程序（即Win32 Console Application），这时产生的应用程序初始没有可视化的界面，需通过OpenGL的相关函数创建可视化窗口以及窗口内的操作；第二种方法和本书第2章创建MFC AppWizard(exe)应用程序的步骤相同，首先建立一个空的可视化应用程序，然后将该程序进一步配置为OpenGL开发环境。由于第二种使用方法创建的应用程序界面比较友好，交互操作方便，所以，下面主要讲述第二种情况下的OpenGL配置方法。

假设创建的应用程序名称为OpenGL001，首先，在GLTest004View.h头文件中加入：

```
#define GLUT_DISABLE_ATEXIT_HACK
#include <gl/gl.h>                    //核心库
#include <gl/glu.h>                   //实用库
#include <gl/glut.h>                  //实用库
#include <gl/glaux.h>                 //辅助库
```

然后，在OpenGL001View.cpp源文件中加入：

```
#pragma comment(lib,"glaux")
#pragma comment(lib,"glut")
#pragma comment(lib,"glu32")
```

在OpenGL001View.cpp文件的成员函数PreCreatWindow()中，加上如下代码：

```
cs.style |= WS_CLIPSIBLINGS | WS_CLIPCHILDREN;
```

用于设置OpenGL绘图窗口的风格。

在COpenGL001View类中，增加如下对应消息的处理函数：

WM_CREATE——对应消息的处理函数：OnCreate()

WM_ERASEBKGND——对应消息的处理函数：OnEraseBkgnd()

WM_SIZE——对应消息的处理函数：OnSize()

WM_DESTROY——对应消息的处理函数：OnDestroy()

Windows的绘图操作在显示设备环境（device context，DC）中进行，OpenGL是在一个称为渲染环境（rendering context，RC）的数据结构中进行，因此，需要将Windows的设备环境替换为OpenGL的渲染环境，首先在GLTest004View.h的View类中增加两个成员：

```
HGLRC m_hRC;                          //渲染环境句柄
CDC * m_pDC;                          //设备指针
```

再增加两个初始化成员函数：

```
BOOL InitializeOpenGL();                        //OpenGL 初始化
BOOL SetupPixelFormat();                        //设置像素格式
```

在 OpenGL001View. cpp 文件中增加两个初始成员函数的实现代码：

```
BOOL COpenGL001View::InitializeOpenGL(){
    //获得视图窗口的设备环境指针
    m_pDC = new CClientDC(this);
    if(m_pDC == NULL){
        MessageBox("获得视图窗口的设备环境失败!");
        return FALSE;
    }
    if(!SetupPixelFormat()) {                   //设置 OpenGL 所需的像素格式
        return FALSE;
    }
    //创建渲染环境句柄
    m_hRC = ::wglCreateContext (m_pDC->GetSafeHdc ());
    if(m_hRC == 0) {
        MessageBox("创建渲染环境失败!");
        return FALSE;
    }
    //将 RC 与 DC 关联起来
    if(::wglMakeCurrent (m_pDC->GetSafeHdc (), m_hRC) == FALSE) {
        MessageBox("RC 与 DC 关联失败!");
        return FALSE;
    }
    return TRUE;
}
```

设置 OpenGL 像素格式的函数代码如下：

```
BOOL COpenGL001View::SetupPixelFormat() {
     static PIXELFORMATDESCRIPTOR pfd = {
         sizeof(PIXELFORMATDESCRIPTOR),         //结构大小
             1,                                 //版本号
             PFD_DRAW_TO_WINDOW |               //支持 Windows
             PFD_SUPPORT_OPENGL |               //支持 OpenGL
             PFD_DOUBLEBUFFER,                  //双缓冲
             PFD_TYPE_RGBA,                     //指定像素类型
             24,                                //颜色深度缓冲区
             0, 0, 0, 0, 0, 0,                  //忽略颜色位
             0,                                 //忽略 Alpha 缓冲
             0,                                 //忽略 Shift 位
             0,                                 //无累积缓冲
             0, 0, 0, 0,                        //无累积缓冲
             16,                                //16 位 z 缓冲
             0,                                 //保留恒为 0
             0,                                 //保留恒为 0
             PFD_MAIN_PLANE,                    //保留恒值
             0,                                 //保留恒为 0
             0, 0, 0                            //保留恒为 0
     };
     int m_nPixelFormat = ::ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd);
```

```
        if (m_nPixelFormat == 0 ) {
            return FALSE;
            }
        if ( ::SetPixelFormat(m_pDC->GetSafeHdc(), m_nPixelFormat, &pfd) == FALSE) {
            return FALSE;
        }
        return TRUE;
}
```

在 OnCreate()中通过调用 InitializeOpenGL()完成 OpenGL 的初始化:

```
int COpenGL001View::OnCreate(LPCREATESTRUCT lpCreateStruct) {
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    InitializeOpenGL();
    return 0;
}
```

在视图窗口的背景重绘函数 OnEraseBkgnd()中,设置不重绘,直接返回,由 OpenGL 设置。

```
BOOL COpenGL001View::OnEraseBkgnd(CDC* pDC) {
//  return CView::OnEraseBkgnd(pDC);        //不重绘背景
        return TRUE;                        //直接返回
}
```

在退出应用程序时,需要释放程序启动和初始化时申请的环境句柄和设备指针,这些操作在 OnDestroy()中完成:

```
void COpenGL001View::OnDestroy() {
    CView::OnDestroy();
    //设置 RC 和 DC 不再关联
    if(::wglMakeCurrent (0,0) == FALSE)  {
        MessageBox("设置 RC 和 DC 不再关联失败!");
    }
    //删除渲染环境句柄
    if(::wglDeleteContext (m_hRC) == FALSE)  {
        MessageBox("渲染环境句柄删除失败!");
    }
    //删除设备指针
    if(m_pDC)  {
        delete m_pDC;
    }
    m_pDC = NULL;
}
```

OpenGL 的坐标系和显示屏幕的像素坐标系不同。OpenGL 采用的是真实环境下的坐标系,显示在屏幕上的图形是对生成图形的真实投影,所以需要将显示屏幕的坐标系转换为 OpenGL 使用的坐标系。坐标系转换在视图类的 OnSize()函数中设置,当窗口尺寸发生变化时,也需要重新进行坐标转换。

在 OnSize()函数中,首先调用 glViewport(0, 0, cx, cy)命令将 OpenGL 的绘图窗口大

小设置成视图窗口的大小，然后调用 glMatrixMode(GL_PROJECTION)命令将当前矩阵设置为投影模式，并调用 glLoadIdentity()命令对矩阵进行初始化。glMatrixMode()函数中的参数可为下面三个枚举常量之一：

GL_PROJECTION　将当前矩阵运算设置为投影矩阵模式

GL_MODELVIEW　将当前矩阵运算设置为模型视图矩阵模式

GL_TEXTURE　将当前矩阵运算设置为纹理映射矩阵模式

一般情况下，在绘制图形对象或者对绘制的图形对象进行几何变换时，需将变换矩阵设置成模型视图矩阵模式；当对绘制的图形对象设置某种显示方式时，则需将变换矩阵设置成投影矩阵的模式，在进行纹理映射时，需将变换矩阵设置为纹理映射的模式。因为要设置坐标转换矩阵，所以在 OnSize()中首先将变换矩阵设置成投影矩阵的模式。

绘图窗口的坐标转换矩阵有三种类型，分别是二维正交投影矩阵、三维正交投影矩阵和透视投影矩阵，对应的命令函数分别为：

void gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)——二维正交投影矩阵，主要用于二维图形的绘制，定义了矩形绘图区域，四个参数分别是左、右、下、上边界，也是个裁剪区域，区域以外的图形被忽略。

void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar)——三维正交投影矩阵，用于三维图形绘制，定义了一个立方体投影区域，参数分别是立方体的左、右、下、上、前、后边界。三维正交投影矩阵可以兼容二维正交投影矩阵。

void gluPerspective (GLdouble fovy, GLdoulbe aspect, GLdouble zNear, GLdouble zFar)——三维图形的透视投影变换矩阵，其中，fovy 为视线对投影区域在 y 方向(即上下方向上)可观察的视线角度，aspect 为视线在投影区域最近观察面上的宽度和高度比值，zNear 和 zFar 分别为投影区域的前后边界。

观察物体时，为了获得更翔实的效果，有时并不在 z 方向观察，而是在其他方向和其他角度观察，类似拿着一个相机绕着物体拍照。这时，需要用 gluLookAt()来设置视点，gluLookAt()的命令函数为：

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery,
GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)
```

函数的参数中，第一组 eyex、eyey、eyez 为相机的坐标位置，第二组 centerx、centery、centerz 为相机镜头对准的物体的坐标位置，第三组 upx、upy、upz 为相机向上的方向在坐标中的矢量方向。

坐标转换设置后，还需要调用 glMatrixMode(GL_MODELVIEW)命令将当前矩阵设置为模型视图模式，并调用 glLoadIdentity()命令对矩阵进行初始化，以便绘制图形。

在坐标转换时，为了实现对屏幕点的交互操作，需要记录坐标转换数据，因此在视图类中增加如下变量：

```
GLdouble aspect_ratio;                    //视图窗口宽高比
GLdouble Win_Size;                        //绘图坐标系的短半轴长度
GLfloat winWidth, winHeight;              //记录视图窗口的宽和高(一半)
```

其中,绘图坐标系的短半轴 Win_Size 的初始值在视图类的构造函数中设置,例如设置

```
Win_Size = 6.0;
```

然后在 OnSize( )函数中进行绘图坐标转换,在 OnSize( )函数中也可以完成其他 OpenGL 初始化设置,如设置视图区域的背景颜色,激活深度检测,设置灯光、材质等。

如下代码在 OnSize()中设置一个坐标原点在绘图区域中心的立方体空间,并将绘图背景颜色设置为白色:

```
void COpenGL001View::OnSize(UINT nType, int cx, int cy) {
    CView::OnSize(nType, cx, cy);
    ::glViewport(0, 0, cx, cy);
    ::glMatrixMode(GL_PROJECTION);          //设置为投影模式
    ::glLoadIdentity();
    aspect_ratio = (GLfloat)cx/(GLfloat)cy;
    if(cx <= cy){
        winWidth = Win_Size;
        winHeight = Win_Size/aspect_ratio;
    }
    else    {
        winWidth = Win_Size * aspect_ratio;
        winHeight = Win_Size;
    }
    //设置一个坐标原点在绘图窗口中心的立方体显示区域
    glOrtho( - winWidth,winWidth, - winHeight,winHeight, - Win_Size,Win_Size);
    winWidth = cx/2.0;
    winHeight = cy/2.0;
    ::glMatrixMode(GL_MODELVIEW);           //设置为模型视图模式
    ::glLoadIdentity();
    glClearColor(1.0f,1.0f,1.0f,1.0f);      //设置绘图背景颜色为白色
}
```

当在视图窗口上拾取屏幕点时,需将拾取点转换为 OpenGL 坐标系的点。针对上述 OnSize()设置的绘图坐标系,屏幕点 point 转换为 OpenGL 绘图坐标系点的方法如下:

```
GLdouble Pt_x,Pt_y,Pt_z;
if(aspect_ratio < 1){
    Pt_x = (point.x - this -> winWidth)/winWidth * Win_Size;
    Pt_y = (this -> winHeight - point.y)/winHeight/aspect_ratio * Win_Size;
    Pt_z = 0.0;
}
else {
    Pt_x = (point.x - this -> winWidth)/winWidth * aspect_ratio * Win_Size;
    Pt_y = (this -> winHeight - point.y)/winHeight * Win_Size;
    Pt_z = 0.0;
}
```

图形实时绘制和显示仍然在视图类的 OnDraw( )函数中实现,OpenGL 采用批处理和内存的"帧缓冲区"技术绘制和显示图形,即首先在内存中绘制图形,通过命令执行,然后将内存中绘制好的图形一次性地显示在显示设备上。因此,在 OnDraw( )函数中,首先通过 glClear()命令在内存中开辟一个连续的颜色缓冲区,并清除其所有颜色数据,如有深度检

测，还需清除深度检测数据，然后在缓冲区绘制图形，最后通过 glFlush() 和 SwapBuffers (m_pDC—>GetSafeHdc())命令执行绘制任务，并交换前台和后台缓冲区的内容。其中，参数值 m_pDC—> GetSafeHdc()为前台的设备环境。

为了使代码具有易读性，图形的具体绘制放在单独的视图类函数中实现。例如，定义绘制体图形的函数名为 RenderScene()，这样 OnDraw()函数中的代码可如下实现：

```
void COpenGL001View::OnDraw(CDC * pDC){
    COpenGL001Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    glClear( GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT);  //清除缓冲区
    RenderScene();                                       //具体绘图函数
    glFinish();                                          //执行绘图函数
    SwapBuffers( m_pDC - > GetSafeHdc() );               //将缓冲区内容输出给设备环境
}
```

绘制具体图形时，只需在 RenderScene()函数中实现即可。例如，画一条简单的直线的代码如下：

```
void COpenGL001View::RenderScene() {
    glColor3f(1.0,0.0,0.0);              //设置绘制图形的颜色
    glLineWidth(4.0);                    //设置线宽,可不设置,默认线宽 1.0
    glBegin(GL_LINES);                   //启动直线绘制命令
        glVertex2i( - 2.0, - 2.0);       //直线起点
        glVertex2i(2.0,2.0);             //直线末点
    glEnd();                             //绘制结束
}
```

执行程序，即可绘制一条红色的有一定线宽的直线。

10.1.2 OpenGL 相关规范介绍

OpenGL 有严格的规范，在使用 OpenGL API 编程时应遵循这些规范，并理解相关流程，以保证程序的可靠性、可移植性和运行的正确性。

首先是数据类型。OpenGL 和 C++的数据类型很类似，但是，OpenGL 的数据类型以 GL 开头，以便区分和辨识。常用的 OpenGL 数据类型如表 10.1-1 所示。

表 10.1-1 OpenGL 数据类型

数据类型	位数	后缀	说明及对应 C++类型
GLint	32	i	有符号整数类型，int
GLfloat	32	f	实数，float
GLdouble	64	d	双精度实数，double
GLshort	16	s	有符号短整数
GLbyte	8	b	有符号字符，char
GLboolean	1		布尔型，值 GL_TRUE、GL_FALSE，0、1，boolean
GLenum	32		枚举型
GLvoid	16		void 指针

表 10.1-1 中的“后缀”列所列出的字符，表示在某个命令函数中要求其参数是某一特定数据类型时，该函数具有相应的后缀。例如，glColor3f()要求参数是 GLfloat 类型，glVertex2i()要求参数是 GLint 整型。

OpenGL 的函数命令分别放在 OpenGL 的核心库（在 gl.h 头文件声明）、实用库（在 glu.h 头文件声明）以及辅助库（在 glaux.h 头文件声明）中。核心库中的函数均以 gl 作为前缀，实用库中的函数以 glu 作为前缀，辅助库中的函数均以 aux 作为前缀。

对于同一个操作函数，当要求的参数个数和数据类型不同时，该函数还可以加特定的后缀，参数的数据类型用相关字母表示，对于参数的个数在后缀中用数字表示，这时函数参数的数量和类型必须遵守后缀设定的要求，如 glColor3f(1.0,0.0,0.0)、glVertex2i(−2,2)。当有 v 后缀时，表示参数应当是一个向量的形式，在 OpenGL 中，向量数据用数组存放。当不含 v 后缀时，各参数为独立的变量。

OpenGL 是顺序执行程序的方式，采用的是状态机制，在绘制图形过程中，当设置某种绘图状态后，就在这个状态下运行，直到程序改变了该状态为止。例如，画线前设置线的颜色和线宽等。当没有专门设置状态时，程序采用的是默认的状态。

OpenGL 内部存在各种运行的状态变量的值，又称模式，使用时利用 glEnable()命令启用，停止时利用 glEnd()命令禁用。

应用程序可以在任何时候查询每个状态变量的当前值，OpenGL 提供的查询命令有 6 个，前 5 个以 glGet 开头：glGetBooleanv()、glGetDoublev()、glGetFloatv()、glGetIntegerv、glGetPointerv()和 glIsEnabled()。具体使用哪个命令取决于希望以何种数据类型返回查询结果。对于某些状态变量，还可以使用更为具体的查询命令。

应用程序还可以利用 glPushAttrib()命令和 glPushClientAttrib()命令，将一组状态变量压入堆栈，对它们进行临时修改，完成任务后，再利用 glPopAttrib()和 glPopClientAttrib()命令将原先压入堆栈的状态弹出堆栈，恢复到原来的模式。例如，在对某图形对象进行几何变换时，首先利用 glPushMatrix()命令将当前坐标矩阵压入堆栈，执行几何变换后，再利用 glPopMatrix()命令弹出堆栈，恢复原来的坐标系状态。

10.2 基本图形及真实感图形绘制

10.2.1 基本图形绘制

OpenGL 支持绘制的基本图形元素有点、线段、折线段、三角形、四边形以及多边形等，利用这些基本图形元素就可以形成复杂的几何图形。

在绘制图形时，可以先设置图形运行状态，然后调用命令 glBegin(GLenum mode)，设置开始输入图形数据，glBegin()命令的参数为要绘制的图形类型，然后输入图形的顶点及其属性，最后以 glEnd()命令结束图形数据输入，通过上述过程即完成一个基本图形的绘制。其中，glBegin()函数的参数 mode 的值为表 10.2-1 所示的枚举型常量。

表 10.2-1　绘制基本图元的枚举常量

图元枚举常量	意　　义
GL_POINTS	绘制点
GL_LINES	绘制线段，可连续多段
GL_LINE_STRIP	绘制折线（多条线段首尾相接）
GL_LINE_LOOP	绘制封闭折线（多条线段首尾相接且首段和末段线段也首尾相接）
GL_TRIANGLES	绘制三角形
GL_TRIANGLE_STRIP	绘制连续的三角形
GL_TRIANGLE_FAN	绘制三角扇形
GL_QUADS	绘制四边形
GL_QUAD_STRIP	绘制连续四边形
GL_POLYGON	绘制多边形

无论图形多么复杂，OpenGL 都是通过 glVertex()函数给出的一个个图形顶点的坐标来实现图形绘制的。glVertex()函数原型为：

```
void glVertexnt(params);
```

它是一个函数族，包含了 24 个函数，函数的参数为顶点的每个坐标数值或者坐标数值的数组，原型中的 n 表示参数个数，可以是 2、3 或 4，而 t 表示参数的数据类型，其中：i/iv——整数/整数数组，f/fv——实数/实数数组，d/dv——双精度实数/双精度实数数组，s/sv——短整数/短整数数组。

1. 图形颜色设置

在绘制整个图形前或者在给出图形顶点数据的过程中都可以通过 glColor()命令设置图形的颜色，该命令的函数原型为：

```
void glColornt(params);
```

该命令与 glVertexnt()相似，也是一个函数族，包含 32 个函数，其中的 n 也是指参数个数，但只能为 3 或 4，因为颜色参数为 R、G、B 或者 R、G、B 和 A，其中 A 为表示颜色透明度的 Alpha 值。t 除了具有与 glVertexnt()相同的数据类型外，还多了一个选项 u，用来修饰 i 和 s，表示无符号整数和无符号短整数，另外还提供了一种 ub 类型，表示无符号字节型。当参数类型为整数(i、s、b 及对应无符号数)时，其取值范围为 0～255；而当参数类型为实数(f 和 d)时，其取值为 0～1 之间的一个小数值。

在程序中利用 glColor()命令设置颜色后，后续的图形即按照设定的颜色绘制，直到又设置新颜色，如果前后两种颜色分别绘制的是同一个图形的两个顶点，那么在两顶点之间的部分按照渐变的过渡颜色绘制。代码如下所示：

```
glColor3f(1.0,0.0,0.0);                     //设置红色
glBegin(GL_LINES);
    glVertex2i(-2,-2);                      //开始绘制第一条直线
    glVertex2i(2,2);
    glVertex2i(-2,2);                       //开始绘制第二条直线
    glColor3f(0.0,0.0,1.0);                 //设置蓝色
    glVertex2i(2,-2);
glEnd();
```

上述代码在绘制第一条直线时，按照设定的红色绘制，第二条直线的第一个点仍然按照红色绘制，第二个点按照新设置的蓝色绘制，两点之间的线段将是从红到蓝的过渡颜色，线段将是一个渐变色线段。

2. 点大小、线宽及线型设置

在 OpenGL 中点有大小，线有宽度，并且与计算机的操作系统和硬件有关，应该根据当前的计算机支持的大小范围设置正确的尺寸，这时使用 glGet()命令获得支持的具体数值，该命令的原型有：

```
void glGetFloatv(GLenum pname,GLfloat * params);
void glGetBooleanv(GLenum pname,GLboolean * params);
void glGetDoublev(GLenum pname,GLdouble * params);
void glGetIntegerv(GLenum pname,GLint * params);
```

其中，参数 pname 为枚举型常量，与点和线有关的常量如表 10.2-2 所示。

表 10.2-2 点和线设置 glGet()有关常量

常 量	意 义
GL_POINT_SIZE	返回当前点的大小
GL_POINT_SIZE_RANGE	返回点的尺寸范围，即点的最大最小尺寸
GL_POINT_SIZE_GRANULARITY	返回点的增量步长
GL_POINT_SMOOTH	返回当前是否支持点的反走样
GL_LINE_WIDTH	返回当前线宽
GL_LINE_WIDTH_RANGE	返回线的尺寸范围，即线的最大最小尺寸
GL_LINE_WIDTH_GRANULARITY	返回线的增量步长
GL_LINE_SMOOTH	返回当前是否支持线的反走样

需要注意的是，使用 GL_POINT_SIZE_RANGE 和 GL_LINE_WIDTH_RANGE 时，返回两个值，因此，需要用数组变量作为命令的第二个参数。glGet()调用后，该数组下标 0 变量存放尺寸范围的最小值，下标 1 变量存放尺寸范围的最大值。

点和线宽的大小在要求的尺寸范围内设置，数字应该是尺寸增量步长的整数倍，然后利用 glPointSize(GLfloat size)命令指定后续点的大小，利用 glLineWidth(GLfloat width)命令指定线宽。它们的尺寸单位都是和绘图尺寸相关的逻辑单位，与像素无关。

关于线型，OpenGL 中除了实线外没有其他线型，如果需要用其他线型绘制图形，需开启直线的点画功能，并调用 glLineStipple()命令设置画线模式。glLineStipple()命令的原型为：

```
viod glLineStipple(GLint factor,GLshort pattern);
```

其中参数 pattern 为画线模式，pattern 值是由 1 或 0 组成的十六进制数，factor 为缩放因子。从这个模式的低位开始，一个像素一个像素地进行处理。如果模式中对应的位是 1，就绘制这个像素，否则就不绘制。模式使用 factor 参数进行扩展，它与 1 和 0 的连续子序列相乘。因此，如果模式中出现了 3 个 1，并且 factor 是 2，那么它们就扩展为 6 个连续的 1。

启用直线点划功能必须以 GL_LINE_STIPPLE 为参数调用 glEnable()命令。绘制完毕，向 glDisable()函数传递同一个参数，从而禁用直线点划功能。如果没有启用点划线功

能，OpenGL 会自动把 pattern 当作 OxFFFF，即－1，并把 factor 当成 1。

3. 多边形填充设置及多边形的网格化细分

在利用 OpenGL 命令绘制三角形、四边形以及多边形时，由于形成了封闭的多边形，OpenGL 在默认状态下会将其内部进行颜色填充，即自动实现多边形的扫描转换，也可以调用 glPolygonMode()命令设置填充方式。glPolygonMode()的原型为：

```
void glPolygonMode(GLenum face,GLenum mode);
```

其中的 face 指设置对应的多边形表面，常量值为：

GL_FRONT——多边形的正面

GL_BACK——多边形的背面

GL_FRONT_AND_BACK——多边形的前后两面

OpenGL 中对正面的规定是这样的：如果多边形的顶点以逆时针顺序出现在屏幕上，则看到的多边形内部的部分为“正面”，即前面，它的反面即为背面，而且要求多边形的边不能自相交。我们可以通过 void glFrontFace(GLenum mode)函数交换图形的正反面。默认情况下，mode 是 GL_CCW，即逆时针为正面；当 mode 是 GL_CW 时顺时针为正面。

glPolygonMode()命令的第二个参数指定填充方式，常量值为：

GL_POINT——绘制构成多边形的顶点

GL_LINE——仅绘制多边形的轮廓线，即不填充

GL_FILL——填充多边形

当不希望填充图形时，调用 glPolygonMode(GL_FRONT_AND_BACK，GL_LINE)命令，后续的绘图操作仅绘出轮廓线，调用 glPolygonMode(GL_FRONT_AND_BACK，GL_FILL)命令，则填充多边形内部。

OpenGL 中认为合法的多边形必须是凸多边形，对于凹多边形以及带内环的等非凸多边形，OpenGL 在填充时会出现不正确的结果。OpenGL 之所以对多边形类型做出限制，是为了更方便地对符合条件的多边形进行快速渲染。凸多边形可被快速地渲染，而复杂多边形难以快速检测出来。

非凸多边形最简单的填充方法是使用 GLU 网格化对象 GLUtesselator 将任意多边形简化为三角形或凸多边形的组合，从而使 OpenGL 能够实现多边形填充。

多边形网格化的基本思路是将所有多边形的顶点坐标发送到网格器 GLUtesselator，然后网格器将多边形网格化，网格化工作完成以后，网格器使用用户定义的回调模式调用实际的 OpenGL 命令来渲染网格化后的多边形。

下面介绍多边形网格化的基本步骤。

第一步，创建一个网格器对象，命令如下：

```
GLUtesselator * tess = gluNewTess();
```

第二步，注册回调函数。在网格化的过程中，多边形网格化后，网格器会调用一系列的回调模式，调用 OpenGL 命令如 glBegin()、glEnd()、glVertex＊()等填充和绘制网格化后的多边形。这些回调函数必须注册，例如，必须注册的三个回调函数为：

```
gluTessCallback(tess,GLU_TESS_BEGIN,(void (CALLBACK * )())&PolyLine3DBegin);
```

```
gluTessCallback(tess,GLU_TESS_VERTEX,(void (CALLBACK * )())&PolyLine3DVertex);
gluTessCallback(tess,GLU_TESS_END,(void (CALLBACK * )())&PolyLine3DEnd);
```

这三个回调函数代码为：

```
void CALLBACK PolyLine3DBegin(GLenum type){
glBegin(type);
}
void CALLBACK PolyLine3DVertex(const GLvoid * data){
 const GLdouble * ptr = (const GLdouble * )data;
    glVertex3dv(ptr);
}
void CALLBACK PolyLine3DEnd(){
glEnd();
}
```

第三步，多边形网格化处理。

首先，调用 gluTessBeginPolygon(tess,NULL)命令，启动网格化。

因为多边形除了必需的外环外可能还有内环，为了区分顶点在哪个环上，在输入每个环的顶点时，首先调用 gluTessBeginContour(tess)，然后，将该环上边的顶点一个一个点地输入，最后一个点会和第一个点自动连接起来，输入的命令函数为：

```
void gluTessVertex(GLUtessellator * tess, GLdouble cords[3], void * vertexData);
```

网格器使用这些顶点坐标执行网格化。所有的顶点需要位于同一个面中。第二个参数是网格化需要的顶点坐标，第三个参数是实际用来渲染的坐标，它不仅是顶点坐标，也可能是颜色坐标、法向量坐标、纹理坐标。例如，顶点坐标数组为 GLdouble quad[][3]，则顶点输入函数为(其中 i 为顶点索引)：

```
gluTessVertex(tess, quad[i],quad[i]);
```

顶点输入完，调用 gluTessEndContour(tess)命令结束该环输入，然后再开始下一个环。所有环输入完成，调用 gluTessEndPolygon(tess)结束输入。

第四步，调用 gluDeleteTess(tess)删除创建的网格器。

需要注意的是，对于边线相交叉的多边形网格化处理有可能会失败，此时不能填充，在绘图时不建议绘制自交叉多边形。

4. 基本图形实现实例

本小节在 10.1 节创建好的应用程序 OpenGL001 的基础上，实现上述 OpenGL 基本图形的绘制，并动态设置绘图区域的坐标长度、图形的颜色、点的大小、线宽、线型以及多边形是否填充。为此，创建一个顶点集合，通过鼠标在屏幕拾取点，实现各种图形绘制。

首先，在 OpenGL001View.h 文件中建立一个三维点的类：

```
class GLPoint{
public:
    GLPoint(){
        x = 0.0;
        y = 0.0;
        z = 0.0;
```

```
    }
public:
    GLfloat x;
    GLfloat y;
    GLfloat z;
};
```

并定义表示各种基本图形绘制的宏常量：

```
#define GLPOINTS            1
#define GLLINES             2
#define GLLINESTRIP         3
#define GLLINELOOP          4
#define GLTRIANGLES         5
#define GLTRIANGLESTRIP     6
#define GLTRIANGLEFAN       7
#define GLQUADS             8
#define GLQUADSTRIP         9
#define GLPOLYGON           10
```

为了创建点的集合变量和后续可能的数学计算，在头文件中加入相关的文件引用：

```
#include <math.h>
#include <afxtempl.h>
```

在 COpenGL001View 类中再加入下面的变量和函数：

```
GLfloat m_iR;                           //图形颜色的红色值
GLfloat m_iG;                           //图形颜色的绿色值
GLfloat m_iB;                           //图形颜色的蓝色值
GLfloat m_iAlpha;                       //颜色透明度
GLfloat m_PtCurSize;                    //点的大小
GLfloat m_LineWidth;                    //线宽
GLshort m_LinePattern;                  //线型
BOOL m_bPolygonFill;                    //是否填充
int m_flag;                             //绘图命令标识
int m_Rflag;                            //是否拾取操作,0: 不,1: 是
CArray<GLPoint,GLPoint> m_Point_Array;  //拾取顶点集合
void InitOperation();                   //绘图操作初始化设置
```

其中，InitOperation()函数为绘制每个图形时把相关参数设置为初始值，代码如下：

```
void COpenGL001View::InitOperation(){
    m_Rflag = 1;
    m_Point_Array.RemoveAll();
    glLoadIdentity();
    Invalidate();
}
```

在类构造函数 COpenGL001View()中设置初值：

```
COpenGL001View::COpenGL001View(){
    Win_Size = 6.0;
```

```
        m_iR = 0;
        m_iG = 0;
        m_iB = 0;
        m_iAlpha = 1.0;
        m_PtCurSize = 1.0;
        m_LineWidth = 1.0;
        m_LinePattern = -1;                    //实线
        m_flag = 0;
        m_Rflag = 0;
        m_bPolygonFill = FALSE;
    }
```

在应用程序的菜单栏和工具栏中分别设置绘制上述各基本图形的ID标识，并通过类向导增加标识符对应的消息映射函数，在其消息映射函数中，增加可以拾取绘制图形顶点的开关标识等代码。例如，在绘制顶点的映射函数中，增加如下代码：

```
void COpenGL001View::OnPoint() {
    m_flag = GLPOINTS;                         //设置开始绘制顶点
    InitOperation();                           //初始化相关参数
}
```

其他图形的操作函数和顶点绘制的代码类似，只是m_flag的值取对应图形的宏常量。

因为需要通过单击拾取顶点和右击停止拾取，所以，在View视图中加入单击和右击的消息映射函数。对应消息映射函数中的代码增加如下：

```
void COpenGL001View::OnLButtonDown(UINT nFlags, CPoint point) {
    if(m_Rflag == 1){//拾取点,并转换为绘图坐标系点
        GLPoint Pt;
        if(aspect_ratio < 1) {
            Pt.x = (point.x - this -> winWidth)/winWidth * Win_Size;
            Pt.y = (this -> winHeight - point.y)/winHeight/aspect_ratio * Win_Size;
            Pt.z = 0.0;
        }
        else {
            Pt.x = (point.x - this -> winWidth)/winWidth * aspect_ratio * Win_Size;
            Pt.y = (this -> winHeight - point.y)/winHeight * Win_Size;
            Pt.z = 0.0;
        }
        m_Point_Array.Add(Pt);
        Invalidate();
    }
    CView::OnLButtonDown(nFlags, point);
}
void COpenGL001View::OnRButtonDown(UINT nFlags, CPoint point) {
    m_Rflag = 0;                               //不再拾取点
    Invalidate();
    CView::OnRButtonDown(nFlags, point);
}
```

由于应用程序也要实现设置绘图区域的坐标长度、图形的颜色、点的大小、线宽、线型以

及多边形是否填充等功能，因此，还需增加对应这些功能的菜单ID标识，并创建对应的消息映射函数。ID标识以及对应的消息映射函数分别为：

ID_COLOR_SET，OnColorSet——设置绘图颜色

ID_POINT_SIZE，OnPointSize——设置点的大小

ID_LINE_WIDTH，OnLineWidth——设置线宽

ID_LINETYPE，OnLinetype——设置线型

ID_POLYGON_FILL，OnPolygonFill——设置是否填充多边形

ID_DRAW_SIZE，OnDrawSize——设置绘图区域的坐标长度

设置绘制的图形颜色时，一种简便的方法是利用颜色对话框来设置颜色。OnColorSet()函数中的代码如下：

```
void COpenGL001View::OnColorSet() {//颜色设置
    CColorDialog dlg;
    dlg.m_cc.Flags| = CC_RGBINIT|CC_FULLOPEN;
    dlg.m_cc.rgbResult = RGB(m_iR * 255,m_iG * 255,m_iB * 255);
    if(IDOK == dlg.DoModal()){
        COLORREF m_clr = dlg.m_cc.rgbResult; //将 dlg.m_cc.rgbResult 获取到的颜色对话框中
                                             //的颜色保存到变量 m_clr 中
        m_iR = GetRValue(m_clr)/255.0;       //颜色中红色的强度值
        m_iG = GetGValue(m_clr)/255.0;       //颜色中绿色的强度值
        m_iB = GetBValue(m_clr)/255.0;       //颜色中蓝色的强度值
        Invalidate();
    }
}
```

当绘制的图形尺寸比较大时，需要动态改变绘图窗口的显示范围，否则有可能造成绘图区域外的图形部分被裁剪掉而不能完整显示的现象，而设置观察图形坐标大小的OnSize()函数只有在屏幕窗口发生变化时才会被调用。为了能够实时改变绘图区域坐标显示范围，可以创建一个输入坐标长度的对话框，如图10.2-1所示，并通过定义的菜单消息映射函数OnDrawSize()来动态修改绘图坐标显示大小。OnDrawSize()的代码和OnSize()中设置坐标系的代码非常类似，具体如下：

```
void COpenGL001View::OnDrawZize() {
    //设置绘图区域短轴的尺寸，相当于把 OnSize()中的坐标转换重新实现一次
    CCoordSetDlg CoordSetDlg;                //输入坐标长度的对话框
    CoordSetDlg.m_WinSize = Win_Size;        //坐标短半轴长度
    if(CoordSetDlg.DoModal()!= IDOK)
        return;
    Win_Size = CoordSetDlg.m_WinSize;
    int cx,cy;
    cx = (int)winWidth * 2;
    cy = (int)winHeight * 2;
    ::glViewport(0, 0, cx, cy);
    ::glMatrixMode(GL_PROJECTION);
    ::glLoadIdentity();
    if(aspect_ratio < 1){
        winWidth = Win_Size;
```

```
        winHeight = Win_Size/aspect_ratio;
    }
    else {
        winWidth = Win_Size * aspect_ratio;
        winHeight = Win_Size;
    }
    glOrtho( - winWidth,winWidth, - winHeight,winHeight, - Win_Size,Win_Size);
    winWidth = cx/2.0;
    winHeight = cy/2.0;
    ::glMatrixMode(GL_MODELVIEW);
    ::glLoadIdentity();
    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    Invalidate();
}
```

设置点的大小和线宽的方法非常类似。首先创建一个设置点大小的对话框，例如，名称为 CPointSizeDlg，如图 10.2-2 所示。



图 10.2-1 坐标长度设置

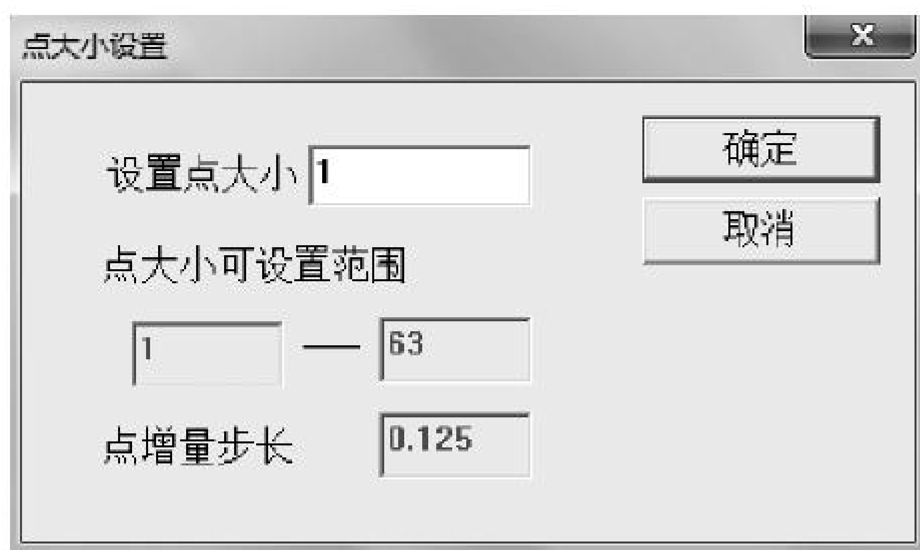


图 10.2-2 点大小设置

对话框中点的大小变量为 m_Size，点变化范围的两个变量为 m_min、m_max，点变动步长为 m_step，则设置点大小的消息函数的代码为：

```
void COpenGL001View::OnPointSize() {
    GLfloat ptSize[2],szStep;
    glGetFloatv(GL_POINT_SIZE_RANGE,ptSize);
    glGetFloatv(GL_POINT_SIZE_GRANULARITY,&szStep);
    CPointSizeDlg PointSizeDlg;
    PointSizeDlg.m_min = ptSize[0];
    PointSizeDlg.m_max = ptSize[1];
    PointSizeDlg.m_step = szStep;
    PointSizeDlg.m_Size = m_PtCurSize;
    if(PointSizeDlg.DoModal() == IDOK){
        m_PtCurSize = PointSizeDlg.m_Size;
        Invalidate();
    }
}
```

同理，对线宽的设置，也是首先创建线宽设置对话框，例如 CLineWidthDlg，如图 10.2-3 所示，对话框中线宽、线宽变动范围以及变动步长的变量分别为 m_Width、m_min、m_max、

m_step。线宽设置的消息函数代码为：

```
void COpenGL001View::OnLineWidth() {
    GLfloat lineSize[2],lineStep;
    glGetFloatv(GL_LINE_WIDTH_RANGE,lineSize);
    glGetFloatv(GL_LINE_WIDTH_GRANULARITY,&lineStep);
    CLineWidthDlg LineWidthDlg;
    LineWidthDlg.m_min = lineSize[0];
    LineWidthDlg.m_max = lineSize[1];
    LineWidthDlg.m_step = lineStep;
    LineWidthDlg.m_Width = this->m_LineWidth;
    if(LineWidthDlg.DoModal() == IDOK){
        m_LineWidth = LineWidthDlg.m_Width;
        Invalidate();
    }
}
```

线型设置，也是首先创建线型对话框，例如 CPatterDlg，如图 10.2-4 所示。除了实线外又设置了另外两种线型，即虚线和中心线，线型选择变量为 m_iPatter。线型设置的函数代码为：

```
void COpenGL001View::OnLinetype() {
    CPatterDlg PatterDlg;
    if(this->m_LinePattern == -1)
        PatterDlg.m_iPatter = 0;
    else if(this->m_LinePattern == 0x3F3F)     //虚线
        PatterDlg.m_iPatter = 1;
    else if(this->m_LinePattern == 0x33FF)     //中心线
        PatterDlg.m_iPatter = 2;
    if(PatterDlg.DoModal() == IDOK){
        if(PatterDlg.m_iPatter == 0)
            this->m_LinePattern = -1;
        else if(PatterDlg.m_iPatter == 1)
            this->m_LinePattern = 0x3F3F;
        else if(PatterDlg.m_iPatter == 2)
            this->m_LinePattern = 0x33FF;
        Invalidate();
    }
}
```



图 10.2-3　线宽设置

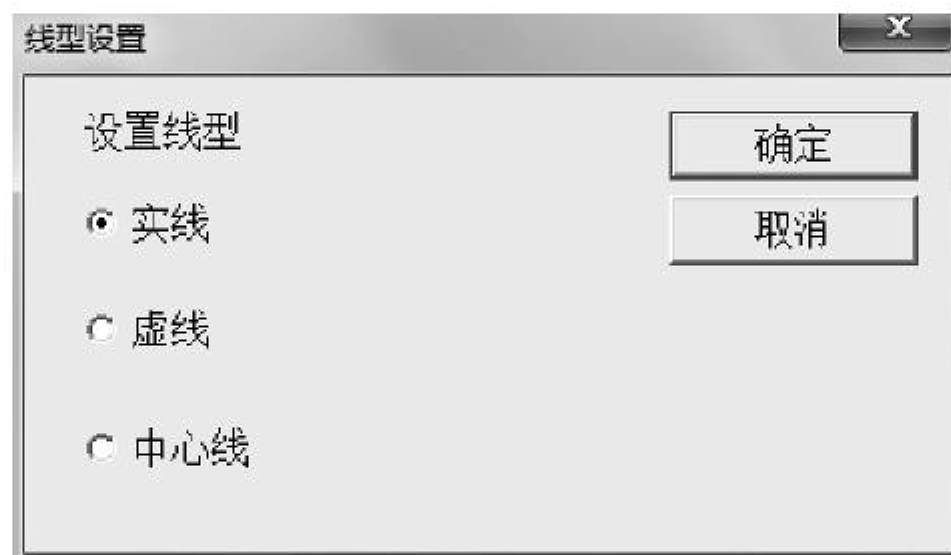


图 10.2-4　线型设置

多边形区域是否填充，创建对话框进行设置，例如 CPolygonFillDlg，如图 10.2-5 所示，是否填充的变量为.m_bFill。函数代码为：

```
void COpenGL001View::OnPolygonFill() {
CPolygonFillDlg PolygonFillDlg;
PolygonFillDlg.m_bFill = m_bPolygonFill;
if(PolygonFillDlg.DoModal() == IDOK){
m_bPolygonFill = PolygonFillDlg.m_bFill;
        Invalidate();
    }
}
```

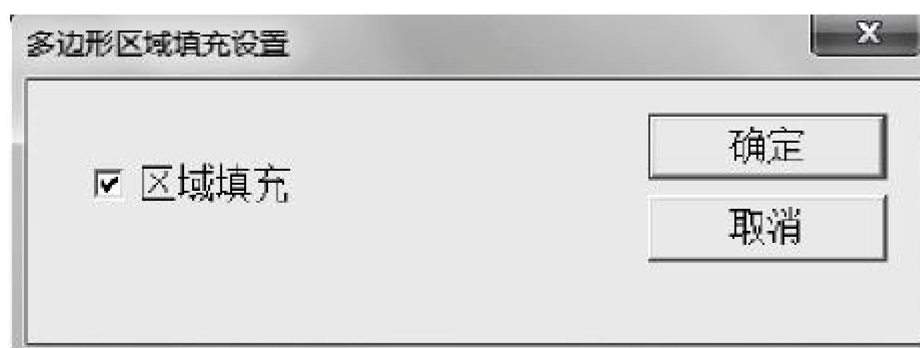


图 10.2-5　多边形填充

在对多边形进行填充时，为使对任意多边形都可正确填充，对多边形进行网络化。网格化的函数代码如下（代码放在 OpenGL001View.cpp 文件或者 OpenGL001View.h 文件中均可）：

```
BOOL PolygonTesselator(CArray<GLPoint,GLPoint>& m_Point_Array){
    int Pt_Num = m_Point_Array.GetSize();          //计算顶点数
    if(Pt_Num<3) return FALSE;
    GLPoint pt;
    GLdouble ( * quad)[3];
    quad = new GLdouble[Pt_Num][3];                //动态设定顶点数组
    for(int i = 0;i<Pt_Num;i++){
        pt = m_Point_Array.GetAt(i);
        quad[i][0] = pt.x;
        quad[i][1] = pt.y;
        quad[i][2] = pt.z;
    }
    GLUtesselator * tess = gluNewTess();           //创建网格器,注册回调函数
gluTessCallback(tess,GLU_TESS_BEGIN,(void (CALLBACK * )())&PolyLine3DBegin);
gluTessCallback(tess,GLU_TESS_VERTEX,(void (CALLBACK * )())&PolyLine3DVertex);
gluTessCallback(tess,GLU_TESS_END,(void (CALLBACK * )())&PolyLine3DEnd);
    gluTessBeginPolygon(tess,NULL);                //启用网格化
        gluTessBeginContour(tess);                 //外环,内环类似
        for(i = 0;i<Pt_Num;i++)
            gluTessVertex(tess, quad[i],quad[i]);     //输入点
        gluTessEndContour(tess);
    gluTessEndPolygon(tess);
    gluDeleteTess(tess);                           //删除网格器
    delete[] quad;
    return TRUE;
}
```

其中三个回调函数(代码在前文已列出,此处不再赘述)也需要放在对应的文件中。通过上述代码设置后,在 RenderScene()函数中绘制和显示图形时,代码如下:

```
void COpenGL001View::RenderScene() {
    glColor3f(m_iR,m_iG,m_iB);                 //设置图形颜色
    if(m_flag == GLPOINTS){                    //绘制点
        glPointSize(this -> m_PtCurSize);
        glBegin(GL_POINTS);
            GLPoint pt;
            for(int i = 0;i < m_Point_Array.GetSize();i++){
                pt = m_Point_Array.GetAt(i);
                glVertex3f(pt.x,pt.y,pt.z);
            }
        glEnd();
    }
    if(m_flag >= 2&&m_flag <= 10){
        //绘制直线
        glLineWidth(m_LineWidth);
        if(m_LinePattern!= - 1){               //设置线性
            glEnable(GL_LINE_STIPPLE);
            glLineStipple(1,m_LinePattern);
        }
        //如是封闭图形,设置是否填充
        if(m_bPolygonFill == TRUE)
            glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
        else
            glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
        if(m_flag == GLLINES)
            glBegin(GL_LINES);
        else if(m_flag == GLLINESTRIP)
            glBegin(GL_LINE_STRIP);
        else if(m_flag == GLLINELOOP)
            glBegin(GL_LINE_LOOP);
        else if(m_flag == GLTRIANGLES)
            glBegin(GL_TRIANGLES);
        else if(m_flag == GLTRIANGLESTRIP)
            glBegin(GL_TRIANGLE_STRIP);
        else if(m_flag == GLTRIANGLEFAN)
            glBegin(GL_TRIANGLE_FAN);
        else if(m_flag == GLQUADS)
            glBegin(GL_QUADS);
        else if(m_flag == GLQUADSTRIP)
            glBegin(GL_QUAD_STRIP);
        else if(m_flag == GLPOLYGON&&m_bPolygonFill!= TRUE)     //不填充时
            glBegin(GL_POLYGON);
                GLPoint pt;
                for(int i = 0;i < m_Point_Array.GetSize();i++){
                    pt = m_Point_Array.GetAt(i);
                    glVertex3f(pt.x,pt.y,pt.z);
                }
```

```
            glEnd();
        if(m_flag == GLPOLYGON&&m_bPolygonFill == TRUE){        //填充时
            PolygonTesselator(m_Point_Array);                   //对多边形网格细分
        }
        if(m_LinePattern!= - 1)
            glDisable(GL_LINE_STIPPLE);
    }
}
```

执行应用程序，可以通过鼠标选取屏幕点，实现 OpenGL 提供的基本图形的绘制。

10.2.2 图形变换与三维绘图

1. 几何变换

为了能够更详细地观察显示图形，通常需要对图形进行几何变换。OpenGL 提供了对平移变换、旋转变换和缩放变换这三种基本变换类型的调用命令函数，对于其他几何变换，OpenGL 使用一个统一的用户定义的变换矩阵调用函数实现。

调用平移变换的命令函数原型为：

```
void glTranslated/f(type x,type y,type z);
```

其中的三个参数 x、y、z 分别为图形在坐标轴 x、y、z 三个方向的平移量。当命令结尾字母为 d 时，各参数的类型为 GLdouble；命令结尾字母为 f 时，各参数的类型为 GLfloat。

调用旋转变换矩阵的命令函数原型为：

```
void glRotated/f(type angle,type x,type y,type z);
```

其中参数 angle 为旋转角度；参数 x、y、z 则生成表示旋转轴的一个向量，例如 x、y、z 为 1、0、0 表示绕 x 轴旋转，x、y、z 为 0、0、1 表示绕 z 轴旋转；命令结尾的 d、f 和平移变换矩阵命令含义相同。

调用缩放变换矩阵的命令函数原型为：

```
void glScaled/f(type x,type y,type z);
```

其参数分别表示在三个坐标轴方面缩放的比例系数。三个值均为正值，其值大于 1 表示将图形在对应坐标方向放大，其值介于 0 和 1 之间表示将图形在对应坐标方向缩小。

对于用户自定义的其他变换矩阵，OpenGL 调用命令函数为：

```
void glMultMatrixf/d(const Type * m);
```

其参数为用户定义的作为变换矩阵的一维数组。该变换矩阵为齐次坐标矩阵。

在 OpenGL 中，矩阵常常是存放在一维数组中而不是二维数组中的，另外，矩阵元素在数组中是按照列的顺序存放，而不是按照行的顺序存放。例如一个 4×4 的矩阵为

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

对应的一维数组为

$$\{a_{11},a_{21},a_{31},a_{41},a_{12},a_{22},a_{32},a_{42},a_{13},a_{23},a_{33},a_{43},a_{14},a_{24},a_{34},a_{44}\}$$

例如，沿 xOz 平面的镜像变换矩阵数组为

```
GLfloat mat[] = {1.0,0.0,0.0,0.0,0.0,-1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,1.0}
```

因为几何变换是图形显示方式矩阵的变化，在变换前，应该保证当前矩阵模式是模型的状态，而且对矩阵格式化开始变换，所以，在 OnSize()函数或者在绘图函数中，通过如下代码设置矩阵的显示模式和最初矩阵状态：

```
glMatrixMode(GL_MODELVIEW);                //GL_MODELVIEW 为显示模型状态
glLoadIdentity();                          //矩阵格式化
```

OpenGL 中的几何变换是将当前的坐标系改变成变换矩阵要求的状态，然后绘制图形，当几何变换完成后，还应将坐标系恢复到最初的状态，坐标恢复原来的状态。除了利用矩阵格式化的方法外，常采用的方法是利用 glPushMatrix()命令将当前坐标矩阵压入堆栈，几何变换执行后，再利用 glPopMatrix()命令弹出堆栈，恢复原来的坐标系状态。

2. 几何变换及图形动画实例

可以通过鼠标、键盘以及动画等方式显示图形变换效果，例如，通过键盘的上、下、左和右箭头键来移动图形，再组合使用 Shift 键实现图形缩放，通过动画实现图形的旋转变换。为此，在应用程序的视图类 COpenGL001View 中增加如下几何变换相关的变量：

```
GLfloat m_lrMove,m_btMove;                 //左右上下移动变量
GLfloat m_rAngle;                          //旋转角度
GLfloat m_Scale;                           //缩放比例
BOOL m_bAnimation;                         //是否采用动画的标识
```

然后，在 COpenGL001View 的构造函数 COpenGL001View()中增加如下初始化代码：

```
m_lrMove = 0.0;
m_btMove = 0.0;
m_rAngle = 0.0;
m_Scale = 1.0;
m_bAnimation = FALSE;
```

同理，在图形操作初始化函数 InitOperation()中也增加和构造函数相同的代码，不过对于动画标识变量的设置，改为如下代码，以便关闭定时器：

```
if(m_bAnimation == TRUE){
    m_bAnimation = FALSE;
    KillTimer(0);
}
```

为了实现键盘操作和动画操作，在应用程序的工具栏中建立一个动画工具栏，ID 标识为 ID_ANIMATION，然后通过类向导，建立键盘消息、定时器以及动画图标 ID_ANIMATION 的消息映射函数：

WM_KEYDOWN——OnKeyDown()

WM_TIMER——OnTimer()

ID_ANIMATION——OnAnimation()

其中,在 OnKeyDown()函数中,当按下上、下、左、右箭头键时设置移动的距离,如果同时按下了 Shift 键,则设置图形可缩放的系数。代码如下:

```
void COpenGL001View::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags) {
    switch (nChar){
        case VK_UP:
            if(GetAsyncKeyState(VK_LSHIFT)&&0X8000)  //Shift 键是否被按下
                m_Scale += 0.1;                      //设置缩放比例
            else
                m_btMove += 0.1;                     //设置上下移动
            break;
        case VK_DOWN:
            if(GetAsyncKeyState(VK_LSHIFT)&&0X8000){
                m_Scale -= 0.1;
                if(m_Scale == 0)
                    m_Scale = 0.05;
            }
            else
                m_btMove -= 0.1;
            break;
        case VK_LEFT:
            m_lrMove -= 0.1;                         //设置水平移动
            break;
        case VK_RIGHT:
            m_lrMove += 0.1;
            break;
        Invalidate();
    }
    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

按下工具栏的动画图标,设置计时器启动,开始动画展示,再按下停止动画展示。代码如下:

```
void COpenGL001View::OnAnimation() {
    if(m_bAnimation == FALSE){
        if(m_flag >= 1&&m_flag <= 10){
            if(m_Point_Array.GetSize()< 2)return;
        }
        SetTimer(0,10,NULL);
        m_bAnimation = TRUE;
    }
    else{
        KillTimer(0);
        m_bAnimation = FALSE;
    }
    Invalidate();
}
```

在计时器中,设置在每一个时间间隔的图形旋转角度,代码如下:

```
void COpenGL001View::OnTimer(UINT nIDEvent) {
    m_rAngle += 1.0;                                    //设置旋转角度
    Invalidate();
    CView::OnTimer(nIDEvent);
}
```

在绘图函数 RenderScene()中，在绘制基本图形前，将原坐标矩阵压入堆栈后，调用平移变换、旋转变换以及缩放变换，然后再绘制图形，最后将原有坐标矩阵顶出堆栈。代码如下。

```
void COpenGL001View::RenderScene() {
    ...//(绘制坐标中心图形,代码此处省略)
    glPushMatrix();                                     //将当前坐标矩阵压入堆栈
    //图形变换
    glTranslatef(m_lrMove,m_btMove,0);                  //平移变换
    glRotatef(m_rAngle,1.0f,0.0f,0.0f);
    glRotatef(m_rAngle,0.0f,1.0f,0.0f);
    glRotatef(m_rAngle,0.0f,0.0f,1.0f);                 //旋转变换
    glScalef(m_Scale,m_Scale,m_Scale);                  //比例变换
        ...//(绘制基本图形,代码此处省略)
    glPopMatrix();                                      //恢复原有坐标矩阵
}
```

运行程序，通过上述代码绘制一个基本图形后，按上、下、左、右箭头键，即可移动图形，同时按 Shift 键和上或下箭头键，实现图形缩放。按下工具栏的动画图标，图形将进行旋转。

3. OpenGL 三维绘图命令

通过上节几何变换的图形实例看出，OpenGL 绘制的实际都是三维图形，只是点、线、面这些基本图形都在一个平面上。OpenGL 核心库没有提供其他任何三维图形对象，不过为了方便编程和构造复杂三维对象，OpenGL 实用库 GLUT 提供了 9 种三维图形对象，在应用程序中可以直接调用生成三维图形，也可以利用它们生成复杂的图形。这 9 种三维图形分别是圆锥体、四面体、正方体、正十二面体、正二十面体、正八面体、球体、圆环体和茶壶，每个图形都有实体图和线框图两种模式，并调用不同的命令进行绘制。其中 glutSolidxxxx()用于绘制实体模式，gluWirexxxx()用于绘制线框模式。

绘制圆锥体的函数原型为：

```
void glutSolidCone(GLdouble base,GLdouble height,GLdouble slices,GLint stacks);
void glutWireCone(GLdouble base,GLdouble height,GLdouble slices,GLint stacks);
```

这两个命令分别绘制底部中心在坐标原点、顶点在 z 轴的实体圆锥面和线框圆锥面，其中，参数 base 为圆锥底面半径，height 为圆锥体的高度，slices 为圆锥体环绕 z 轴的分段数，stacks 为圆锥体沿 z 轴的分段。slices 和 stacks 的大小决定了圆锥面的光滑程度。

绘制四面体的函数原型为：

```
void glutSolidTetrahedron();
void glutWireTetrahedron();
```

这两个命令绘制中心在原点、外接圆的半径是$\sqrt{3}$个单位的四面体实体图和线框图。它们没有带参数，在绘制时，可以首先调用 glScale()命令设置缩放倍数。

绘制正方体的函数原型为：

```
void glutSolidCube(GLdouble size);
void glutWireCube(GLdouble size);
```

这两个命令绘制中心位于原点的实体立方体和线框立方体，其中 size 为立方体的边长。

绘制正十二面体的函数原型为：

```
void glutSolidDodecahedron();
void glutWireDodecahedron();
```

这两个命令绘制中心在原点、外接圆的半径是$\sqrt[2]{3}$个单位的正十二面体实体图和线框图。它们也没有参数，使用方法也是在绘制时首先调用 glScale()命令设置缩放倍数。

绘制正二十面体的函数原型为：

```
void glutSolidIcosahedron();
void glutWireIcosahedron();
```

这两个命令绘制中心在原点、外接圆的半径是 1 的正二十面体实体图和线框图。它们也没有参数，使用方法也是在绘制时首先调用 glScale()命令设置缩放倍数。

绘制正八面体的函数原型为：

```
void glutSolidOctahedron();
void glutWireOctahedron();
```

这两个命令绘制中心在原点、外接圆的半径是 1 的正八面体实体图和线框图。它们也没有参数，使用方法也是在绘制时首先调用 glScale()命令设置缩放倍数。

绘制球体的函数原型为：

```
void glutSolidSphere(GLdouble radius,GLint slices,GLint stacks);
void glutWireSphere(GLdouble radius,GLint slices,GLint stacks);
```

这两个命令绘制球心位于原点、半径为 radius 的球的实体图和线框图。其中，slices 为绕 z 轴的分段数，stacks 为沿 z 轴的分段数。slices 和 stacks 的大小决定了球面的光滑程度。

绘制圆环体的函数原型为：

```
void glutSolidTorus(GLdouble inRad,GLdouble outRad,GLint slices,GLint stacks);
void glutWireTorus(GLdouble inRad,GLdouble outRad,GLint slices,GLint stacks);
```

这两个命令绘制中心位于原点、关于 z 轴对称、内径为 inRad、外径为 outRad 的圆环实体图和线框图。其中，slice 为沿周向的分段数，stacks 为沿径向的分段数。

绘制茶壶的函数原型为：

```
void glutSolidTeapot(GLdouble size);
void glutWireTeapot(GLdouble size);
```

这两个命令绘制中心在原点、壶身外接圆半径为size的茶壶实体图和线框图。需要说明的是茶壶不是基本的三维图形元素，但是它本身构造比较复杂，通过茶壶可以看到OpenGL的三维建模能力。

在应用程序中实现上述的三维图形时，首先在OpenGL001View.h头文件中定义表示各三维图形的宏常量，如下所示：

```
#define CONE          11
#define TETRAHEDRON   12
#define CUBE          13
#define DODECAHEDRON  14
#define ICOSAHEDRON   15
#define OCTAHEDRON    16
#define SPHERE        17
#define TORUS         18
#define TEAPOT19
```

在应用程序的菜单栏或者工具栏中设置绘制各三维图形的ID标识，然后通过类向导建立各个ID标识的消息映射函数，在该函数中设置绘图命令标识m_flag为绘制三维图形的对应宏常量。例如，绘制圆锥体的代码如下，其他类似：

```
void COpenGL001View::OnCone() {
    m_flag = CONE;                                  //设置绘制圆锥体的标识
    InitOperation();
}
```

在绘图函数RenderScene()中，在“glPopMatrix();”代码前，增加如下绘制三维图形的代码(其中，m_3DRadius在COpenGL001View类中定义为GLdouble变量)：

```
if(m_flag == CONE) {                               //圆锥体
    m_3DRadius = Win_Size/4.0;
    if(m_bPolygonFill == TRUE)
        glutSolidCone(m_3DRadius,m_3DRadius * 2,30,30);
    else
        glutWireCone(m_3DRadius,m_3DRadius * 2,30,30);
}
else if(m_flag == TETRAHEDRON){                    //四面体
    m_3DRadius = Win_Size/4.0;
    glScaled(m_3DRadius,m_3DRadius,m_3DRadius);    //缩放变换
    if(m_bPolygonFill == TRUE)
        glutSolidTetrahedron();
    else
        glutWireTetrahedron();
}
else if(m_flag == CUBE){                           //立方体
    m_3DRadius = Win_Size/4.0;
    if(m_bPolygonFill == TRUE)
        glutSolidCube(m_3DRadius);
    else
        glutWireCube(m_3DRadius);
}
```

```
else if(m_flag == DODECAHEDRON){                        //正十二面体
    m_3DRadius = Win_Size/4.0;
    glScaled(m_3DRadius,m_3DRadius,m_3DRadius);
    if(m_bPolygonFill == TRUE)
        glutSolidDodecahedron();
    else
        glutWireDodecahedron();
}
else if(m_flag == ICOSAHEDRON ){                        //正二十面体
    m_3DRadius = Win_Size/4.0;
    glScaled(m_3DRadius,m_3DRadius,m_3DRadius);
    if(m_bPolygonFill == TRUE)
        glutSolidIcosahedron();
    else
        glutWireIcosahedron();
}
else if(m_flag == OCTAHEDRON){                          //正八面体
    m_3DRadius = Win_Size/4.0;
    glScaled(m_3DRadius,m_3DRadius,m_3DRadius);
    if(m_bPolygonFill == TRUE)
        glutSolidOctahedron();
    else
        glutWireOctahedron();
}
else if(m_flag == SPHERE){                              //球体
    m_3DRadius = Win_Size/4.0;
    if(m_bPolygonFill == TRUE)
        glutSolidSphere(m_3DRadius,30,30);
    else
        glutWireSphere(m_3DRadius,30,30);
}
else if(m_flag == TORUS){                               //圆环体
    m_3DRadius = Win_Size/4.0;
    if(m_bPolygonFill == TRUE)
        glutSolidTorus(m_3DRadius/2.0,m_3DRadius,30,30);
    else
        glutWireTorus(m_3DRadius/2.0,m_3DRadius,30,30);
}
else if(m_flag == TEAPOT){                              //茶壶
    m_3DRadius = Win_Size/4.0;
    if(m_bPolygonFill == TRUE)
        glutSolidTeapot(m_3DRadius);
    else
        glutWireTeapot(m_3DRadius);
}
```

通过上述代码,就可以绘制 GLUT 实用库中提供的 9 种类型的三维图形。图 10.2-6 所示为绘制的茶壶线框图和实体图。

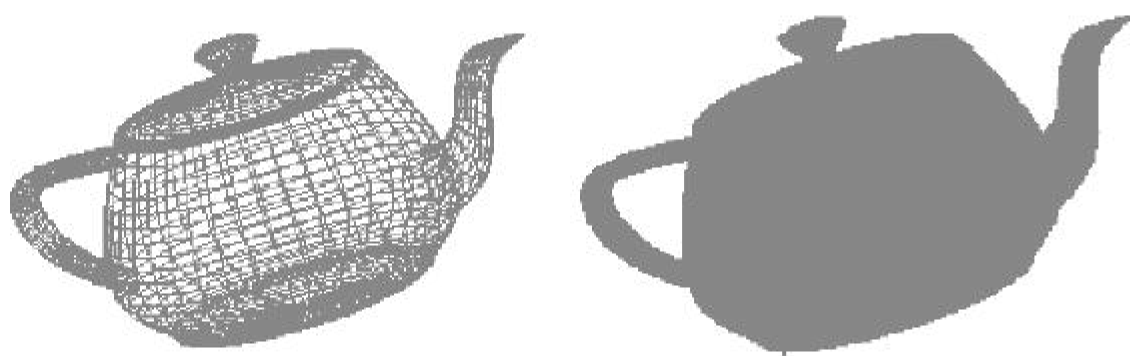


图 10.2-6 茶壶的线框图和实体图

10.2.3 真实感图形显示

图 10.2-6 中显示的茶壶三维图形存在二义性以及无立体感的问题，原因是图形没有进行消隐和光照等真实感显示处理。OpenGL 在绘制真实感图形时，需要通过设置深度检测、设置场景的光照以及材质等方式使图形达到真实感的效果，这些设置只对 OpenGL 提供的三维图形实体图以及指定了正确的表面法向量的平面填充图形有效。

1. 深度检测

所谓深度检测，是对组成三维实体的各表面与视点的距离进行检测，这是实现消隐的第一步。深度检测也是一个状态，在使用前调用 glEnable(GL_DEPTH_TEST)命令启用该状态，如果不启用，则利用 glDisable(GL_DEPTH_TEST)命令关闭该状态。当启用了深度检测状态后，在显示图形的 OnDraw()函数中，在具体绘图函数 RenderScene()之前要调用 glClear(GL_DEPTH_BUFFER_BIT)命令，以清除深度缓冲区。

2. 光照

对于真实感图形显示，设置光照是极为重要的一个步骤，如果没有光照，绘制的图形将如图 10.2-6 所示无立体感。为了真实地表现出现实世界的景象，OpenGL 为三维场景的绘制提供了多种光源，这些光源以灯光的形式出现，OpenGL 允许一个场景至少支持 8 盏灯。OpenGL 中的光源主要有环境光、漫反射光、镜面反射光以及自发光四种类型，不同的光源具有不同的属性，在设置光源时需要采用不同的参数。其中前面三种光源在本书前文已经论述，此处不再赘述。自发光本身可以作为光源，它的材质是一种光照属性，使其表面亮度高一些，例如场景中有一个开着的灯泡、电棒或者车灯，要显示它们是发光的样子。

设置灯光的第一步是启用光照，光照也是 OpenGL 的一种状态，因此，也要调用 glEnable()命令。具体命令为 glEnable(GL_LIGHTING)，启用光照状态，然后，调用 glLight()命令设置光源类型以及灯光位置，该命令原型为：

```
Void glLightf/iv(GLenum light,GLenum pname,const GLfloat * params);
```

其中 light 为灯光的序号，参数形式为 GL_LIGHTi，$0 \leqslant i < GL_MAX_LIGHTS$，GL_MAX_LIGHTS 的具体值与设备有关，至少为 OpenGL 可支持的灯盏数 8。参数 pname 为一个设置灯光某一属性的枚举常量，可为如表 10.2-3 所示的常量值。

参数 params 为一数组，表示光源时为 R、G、B 和 Alpha 分量的 4 个值，表示光源位置时为齐次坐标的 4 个值。

某盏灯设置后，如果要将该盏灯打开，需调用 glEnable(GL_LIGHTi)命令，i 为该灯序号；关闭该灯调用 glDisable(GL_LIGHTi)命令。当场景中有多盏灯时，可以根据需要随时

利用 glEnable()和 glDisable()命令打开和关闭某盏灯。

表 10.2-3 参数 pname 的枚举常量

枚举常量	意义
GL_AMBIENT	环境光
GL_DIFFUSE	漫反射光
GL_SPECULAR	镜面光或聚光灯
GL_POSITION	灯光位置
GL_SPOT_DIRECTION	聚光灯投射方向
GL_SPOT_EXPONENT	聚光灯指数(单一值)
GL_SPOT_CUTOFF	聚光灯投射角度(单一值)
GL_CONSTANT_ATTENUATION GL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	灯光的衰减因子,分别为常数(无衰减)、线性误差和平方误差(单一值)

3. 材质

在绘制三维图形时,除了设置光照外,还需要设置图形对象的材质,来表现其对光照的反应即色调。若不进行设置,最终绘出的图形对象颜色取决于绘图颜色和光源颜色。

首先,调用 glEnable(GL_COLOR_MATERIAL)命令启用材料对当前绘图颜色的跟踪,然后,调用 glColorMaterial()命令指定哪个面对光源跟踪,该命令原型为:

```
void glColorMaterial(GLenum face,GLenum mode);
```

其中,参数 face 用来指定对象的跟踪表面,为枚举型常量,常量可为 GL_FRONT(前面)、GL_BACK(后面)或者 GL_FRONT_AND_BACK(前后两面);mode 用来指定跟踪什么光源,为枚举型常量,常量可为下列之一:GL_EMISSION(自发光)、GL_AMBIENT(跟踪环境光)、GL_DIFFUSE(跟踪漫反射光)、GL_SPECULAR(跟踪聚光灯)和 GL_AMBIENT_AND_DIFFUSE(跟踪环境光和漫反射光)。

在光照中如果设置了镜面光或者聚光灯等,那么在设置材质参数时,对于表面的高光部分,还需调用 glMaterial()命令设置高光反射特性。例如,设置某号灯盏具有聚光灯特性:

```
GLfloat specularLight[] = {1.0f,1.0f,1.0f,1.0f};
glLightfv(GL_LIGHT0,GL_SPECULAR,specularLight);
```

那么,设置表面对高光的反射特性和反射系数:

```
glMaterialfv(GL_FRONT,GL_SPECULAR,specularLight);
glMateriali(GL_FRONT,GL_SHININESS,100);
```

glMarerial()命令的函数原型为:

```
void glMaterialX(GLenum face,GLenum pname,Type params)
```

其中,X 表示参数的数据类型,可以为 f、i、fv 以及 iv;参数 face 指定设置材质的对象表面;枚举常量和 glColorMaterial()中表面使用的枚举常量相同。pname 用来指定设置材质的哪些特性参数,所用的枚举常量如表 10.2-4 所示。

表 10.2-4 参数 pname 的枚举常量

枚 举 常 量	意 义
GL_AMBIENT	环境光
GL_DIFFUSE	漫反射光
GL_SPECULAR	光或聚光灯
GL_EMISSION	自发光
GL_SHININESS	反射系数,单一值,取值[0,128]
GL_AMBIENT_AND_DIFFUSE	环境光与漫反射光
GL_COLOR_INDEXES	索引色,仅当绘图色为索引色时可用

OpenGL 对光照处理有一个设置光照模型参数的命令:

```
void glLightModelX(GLenum pname,Type param);
```

其中,X 表示参数的数据类型,可以为 f、i、fv 以及 iv。pname 为枚举常量,可为以下值。

GL_LIGHT_MODEL_LOCAL_VIEWER——指出计算镜面反射的方式。若参数 param 值为 0,则镜面反射角与视线方向平行且指向 z 轴反方向;否则,镜面反射角从眼睛坐标系的原点计算。

GL_LIGHT_MODEL_TWO_SIDE——若参数 param 的值为 0,则为单面光照,光照计算仅与材质的前面有关,否则,为双面光照。

GL_LIGHT_MODEL_AMBIENT——参数 param 为 4 个数据组成的向量,它给出了环境光中 RGBA 的强度。

需要注意的是,当设置了光照模型参数后,一般情况下,要关闭 GL_COLOR_MATERIAL 状态。

4. 表面法向量以及正反面

在绘制三维实体图形时,需要为构成三维实体的各面指定法向量。OpenGL 中所有实体表面均由平面构成,即使曲面也是利用平面逼近的。在绘制实体表面的平面时,除了用顶点向量表示平面边界外,平面法向量也是一个极为重要的参数,它为识别平面的正反面以及光照处理提供了依据。

OpenGL 的表面法向量和计算机图形学中的表面外法线向量相同,也是从实体表面的正面垂直指向外部的方向矢量。绘制平面图形时,在给定顶点向量之前,首先调用 glNormal()命令指定该表面的法向量,该命令原型为:

```
void glNormal3X(Type nx,Type ny,Type nz);
void glNormalX(const Type * v);
```

其中,X 为 b、d、f、i、s 以及 bv、dv、fv、iv、sv 之一,nx、ny、nz 参数为 3 个独立的字节型、双精度型、浮点型、整型以及短整型数据,v 是这些类型的数组。

应当说明的是,参数所构成的向量必须是一个单位向量,即它们的模为单位 1,否则必须进行规格化处理为单位向量。当三维图形通过缩放等图形变换后,有可能会导致表面的法向量不再为单位化,这时,可以调用 glEnable(GL_NOMALIZE)命令对法向量重新单位化,从而保证渲染效果。

表面正反面对计算平面法向量有重要作用,在 OpenGL 默认设置情况下,正面看一个

封闭的多边形，如果该多边形顶点走向是逆时针，则正对的多边形表面是正面，反面为背面。如果要求多边形顶点走向为顺时针时，正对的多边形表面是正面，则需要调用 glFrontFace(GL_CW)命令进行设置。

5. 真实感图形显示实例

本节对前文绘制的 OpenGL 三维图形设置深度检测、光照以及材质等真实感特性，来观察显示效果。首先，在 View 视图类中设置如下和真实感显示相关的开关变量和函数：

```
BOOL m_DepthFlag;                  //深度检测设置
BOOL m_LightFlag;                  //启用光照状态
BOOL m_Light0Flag;                 //0 号灯状态
BOOL m_Light1Flag;                 //1 号灯状态
BOOL m_LithtModelFlag;             //是否启用光照模型
BOOL m_MaterialColorFlag;          //是否跟踪当前绘图颜色
BOOL m_MatEmissionFlag;            //设置材质是否自发光
//设置背景颜色
GLfloat m_iR_BG;
GLfloat m_iG_BG;
GLfloat m_iB_BG;
void RealEnvimentSet();            //真实感设置函数
```

在视图类的构造函数 COpenGL001View()中对上述变量设初始值：

```
m_DepthFlag = FALSE;               //深度检测设置
m_LightFlag = FALSE;               //启用光照状态
m_Light0Flag = FALSE;              //0 号灯状态
m_Light1Flag = FALSE;              //1 号灯状态
m_LithtModelFlag = FALSE;          //是否启用光照模型,如启用,则材质不再跟踪当前绘图颜色
m_MaterialColorFlag = FALSE;       //是否跟踪当前绘图颜色
m_MatEmissionFlag = FALSE;         //设置材质自发光
m_iR_BG = 1.0;                     //背景颜色
m_iG_BG = 1.0;
m_iB_BG = 1.0;
```

真实感图形设置函数 RealEnvimentSet()代码为：

```
void COpenGL001View::RealEnvimentSet(){
    if(m_DepthFlag == TRUE)        //深度检测设置
        glEnable(GL_DEPTH_TEST);
    else
        glDisable(GL_DEPTH_TEST);
    if(m_LightFlag == TRUE)        //启用光照状态
        glEnable(GL_LIGHTING);
    else
        glDisable(GL_LIGHTING);
    //设置灯光
    GLfloat ambientLight[] = {0.1f,0.1f,0.1f,1.0f};
    GLfloat diffuseLight[] = {0.9f,0.9f,0.9f,1.0f};
    GLfloat specularLight[] = {1.0f,1.0f,1.0f,1.0f};
    GLfloat light0Pos[] = { - Win_Size, - Win_Size, - Win_Size,1.0f};
    GLfloat light1Pos[] = {Win_Size,Win_Size,Win_Size,1.0f};
```

```
    //0 号灯光
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specularLight);
    glLightfv(GL_LIGHT0,GL_POSITION,light0Pos);
    if(m_Light0Flag == TRUE)        //0 号灯状态
        glEnable(GL_LIGHT0);
    else
        glDisable(GL_LIGHT0);
    //1 号灯光
    glLightfv(GL_LIGHT1,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT1,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT1,GL_SPECULAR,specularLight);
    glLightfv(GL_LIGHT1,GL_POSITION,light1Pos);
    if(m_Light1Flag == TRUE)        //1 号灯状态
        glEnable(GL_LIGHT1);
    else
        glDisable(GL_LIGHT1);
    //材质设置
    GLfloat mat_ambient[] = {0.2f,0.2f,0.2f,1.0};          //材质正面对环境光的反射
    GLfloat mat_diffuse[] = {0.1f,0.5f,0.6f,1.0f};         //材质正面对漫反射光的反射
    GLfloat mat_specular[] = {0.5f,0.5f,0.5f,1.0f};        //材质正面对镜面光的反射
    GLfloat mat_back_ambient[] = {0.1f,0.1f,0.1f,1.0};     //材质背面对环境光的反射
    GLfloat mat_back_diffuse[] = {0.1f,0.4f,0.5f,1.0f};    //材质背面对漫反射光的反射
    GLfloat mat_back_specular[] = {0.3f,0.3f,0.3f,1.0f};   //材质背面对镜面光的反射
    if(m_LithtModelFlag == TRUE){           //是否启用光照模型,如启用,材质不再跟踪当前颜色
        glDisable(GL_COLOR_MATERIAL);
        glLightModelf(GL_LIGHT_MODEL_TWO_SIDE,1.0);        //双面光照
        GLfloat lmodel_ambient[] = {0.4f,0.4f,0.4f,1.0f};
        GLfloat local_view[] = {0.0f};
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lmodel_ambient);
        glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER,local_view);

    }
    else{
        if(m_MaterialColorFlag == TRUE){                   //是否跟踪当前绘图颜色
            glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);
        }
        else
            glDisable(GL_COLOR_MATERIAL);
    }
    { //设置材质对各种光的反射
        glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
        glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
        glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
        glMaterialf(GL_FRONT,GL_SHININESS,50.0);
        glMaterialfv(GL_BACK,GL_AMBIENT,mat_back_ambient);
        glMaterialfv(GL_BACK,GL_DIFFUSE,mat_back_diffuse);
        glMaterialfv(GL_BACK,GL_SPECULAR,mat_back_specular);
        if(m_MatEmissionFlag == TRUE){                     //设置材质自发光
```

```
                GLfloat mat_emission[ ] = {0.3f,0.2f,0.2f,1.0f};  //自发光
                glMaterialfv(GL_FRONT,GL_EMISSION,mat_emission);
            }
            else{
                GLfloat mat_emission[ ] = {0.0f,0.0f,0.0f,1.0f};  //自发光
                glMaterialfv(GL_FRONT,GL_EMISSION,mat_emission);
            }
        }
        //设置背景颜色
        glClearColor(m_iR_BG,m_iG_BG,m_iB_BG,1.0f);
    }
```

当应用程序的窗口发生变化或者坐标发生变化时，灯光及材质的位置等都会改变，所以在 OnSize()和 OnDrawZize()函数中都应调用 RealEnvimentSet()设置真实感属性。由于 RealEnvimentSet()函数也设置了绘图的背景色，所以把 OnSize()和 OnDrawZize()中设置背景颜色的命令“glClearColor(1.0f,1.0f,1.0f,1.0f);”去掉。

当设置深度检测后，在 OnDraw()函数中，需将 glClear(GL_COLOR_BUFFER_BIT)修改为 glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT)，以便清除深度缓冲区。

为实现真实感图形各参数的开关设置，可以创建一个非模式对话框，例如 CEnvSetDlg，界面如图 10.2-7 所示。在程序的菜单栏或者工具栏中建立一个图标，打开这个真实感图形设置对话框。

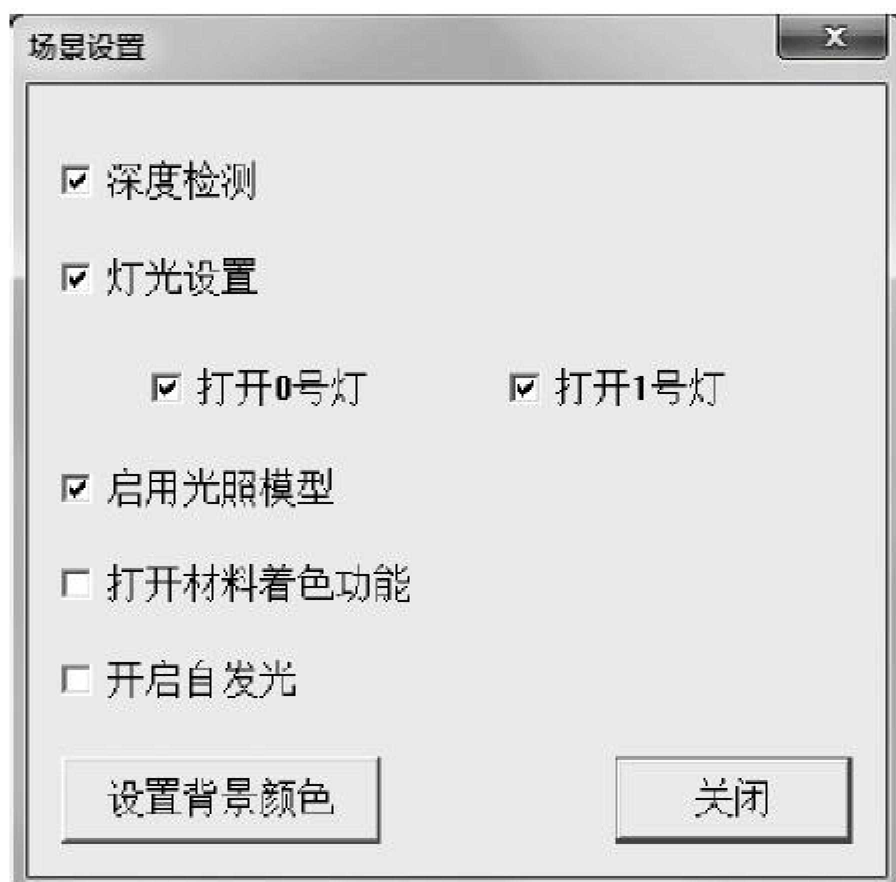


图 10.2-7 真实感图形设置

非模式对话框的创建和使用方法见本书前面章节。

设置对话框里的 CheckBox 复选框名称和真实感图形参数名称对应，并通过类向导为每个复选框创建 BOOL 变量，添加其消息映射函数。

例如，对于深度检测对应的 ID 标识符是 IDC_CHECK_DEPTH，对应的变量为 m_boolDepth，m_bool Depth 的初始值为 FALSE，对应的消息映射函数为 OnCheckDepth()。

当选中对话框的“深度检测”复选框后，调用视图类的真实感设置函数 RealEnvimentSet()，并调用视图刷新函数，实时启用深度检测。函数代码如下：

```
void CEnvSetDlg::OnCheckDepth() {
    UpdateData(TRUE);
    if(this->m_boolDepth == TRUE)
        this->m_pView->m_DepthFlag = TRUE;
    else
        this->m_pView->m_DepthFlag = FALSE;
    this->m_pView->RealEnvimentSet();                    //真实感设置函数
    this->m_pView->Invalidate();                         //实时刷新
    UpdateData(FALSE);
}
```

同理，对话框中其他真实感图形参数设置方法和上述深度检测设置方法类似，代码此处省略。在该对话框中，也可以设置绘图背景的颜色，设置方法和10.2.1节绘图颜色的设置方法类似。

运行应用程序，打开上述的真实感图形参数设置即场景对话框，设置各真实感图形的参数为打开或关闭状态，可以看到绘制图形的显示变化情况。其中，深度检测、灯光设置以及灯的开启直接影响图形的立体感，光照模型、材质着色以及材质自发光等属性使图形的色调进一步发生了变化。

6. 绘制自定义三维拉伸图形

除了立方体、圆锥体、圆环、球、多面体和茶壶等9种三维图形对象的绘图命令外，OpenGL没有提供其他三维图形的绘制方法。对于其他三维形体，可以利用点、线、面这些基本图元来创建三维形体的各表面，将各表面组成一个封闭的形状，即可确定一个三维实体。例如，将一个封闭的平面多边形沿其垂直方向拉伸一定的距离，形成的立体即为三维拉伸体，填充后即为拉伸实体。

为了实现上述自定义的三维拉伸体，首先在OpenGL001View.h头文件中建立拉伸的宏常量：

```
#define STRETCH     20                                   //拉伸
```

在应用程序的视图类COpenGL001View中定义一个多边形的顶点集合变量以及拉伸长度的变量：

```
CArray<GLPoint,GLPoint> m_Pt_Array_Polygon;              //多边形顶点集合
double m_dLength;                                        //拉伸长度
```

然后在应用程序的工具栏中设置拉伸工具栏标识，例如ID_STRETCH，并通过类向导在视图类中建立ID_STRETCH的ON_COMMAND和ON_UPDATE_COMMAND_UI消息映射函数。

要构造三维拉伸体，首先应形成一个封闭的平面多边形，可以通过绘制封闭的折线段实现，并用鼠标在屏幕拾取顶点。**需要注意的是：为保证多边形表面法向量计算正确，鼠标要按逆时针方向拾取屏幕点**。当拾取的顶点达到三个后，拉伸工具条激活，函数代码如下：

```
void COpenGL001View::OnUpdateStretch(CCmdUI* pCmdUI) {
    pCmdUI->Enable((m_flag == GLLINELOOP&&m_Point_Array.GetSize()>= 3)?TRUE:FALSE);
}
```

顶点拾取后，单击拉伸工具条，将拾取点赋给多边形顶点集合，设置拉伸长度，开启拉伸标识。函数代码如下：

```
void COpenGL001View::OnStretch() {
    m_Pt_Array_Polygon.RemoveAll();
    m_Pt_Array_Polygon.Append(this->m_Point_Array);        //将顶点赋给拉伸多边形
    m_dLength = Win_Size/2.0;                              //设置拉伸长度
    m_flag = STRETCH;                                      //设置拉伸标识
    InitOperation();
}
```

在绘图函数 RenderScene()中增加绘制拉伸体的代码，代码放在函数最后的 glPopMatrix()命令之前：

```
if(m_flag == STRETCH){//拉伸体
    if(m_bPolygonFill == TRUE)                          //设置是否填充
        glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    else
        glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    GLfloat ver[3][3],nor[3];
    GLPoint pt,pt1,pt2,pt3;
    //绘制上下两个底,首先绘制下底面,假设顶点是逆时针走向
    if(m_dLength>0)
        glNormal3f(0.0,0.0,-1.0);
    else
        glNormal3f(0.0,0.0,1.0);
    PolygonTesselator(m_Pt_Array_Polygon);          //利用多边形网格化细分多边形
    //绘制上底面
    if(m_dLength>0)
        glNormal3f(0.0,0.0,1.0);
    else
        glNormal3f(0.0,0.0,-1.0);
    CArray<GLPoint,GLPoint> m_Pt_Array_Polygon1;
    for(int i = this->m_Pt_Array_Polygon.GetSize()-1;i>=0;i--){
        pt = m_Pt_Array_Polygon.GetAt(i);
        pt.z += m_dLength;
        m_Pt_Array_Polygon1.Add(pt);
    }
    PolygonTesselator(m_Pt_Array_Polygon1);         //多边形网格化细分多边形
    //绘制侧面,首先计算外法向量
    for(i = 0;i<this->m_Pt_Array_Polygon.GetSize();i++){
        pt = m_Pt_Array_Polygon.GetAt(i);
        if(i<this->m_Pt_Array_Polygon.GetSize()-1)
            pt1 = m_Pt_Array_Polygon.GetAt(i+1);
        else
            pt1 = m_Pt_Array_Polygon.GetAt(0);
        pt2 = pt1;
        pt2.z += m_dLength;
        pt3 = pt;
        pt3.z += m_dLength;
        //构造向量点
```

```
            ver[0][0] = pt.x;ver[0][1] = pt.y;ver[0][2] = pt.z;
            ver[1][0] = pt1.x;ver[1][1] = pt1.y;ver[1][2] = pt1.z;
            ver[2][0] = pt2.x;ver[2][1] = pt2.y;ver[2][2] = pt2.z;
            CaCulateNormal(ver,nor);                    //计算法向量
            glBegin(GL_POLYGON);
                glNormal3fv(nor);                       //设置法向量
                glVertex3f(pt.x,pt.y,pt.z);
                glVertex3f(pt1.x,pt1.y,pt1.z);
                glVertex3f(pt2.x,pt2.y,pt2.z);
                glVertex3f(pt3.x,pt3.y,pt3.z);
            glEnd();
        }
    }
```

上述函数代码中，在绘制每个表面多边形时，需要首先设置表面的法向量。对于上下两个底面多边形的外法线向量，根据拉伸长度的正负，设置上下两个底面多边形的外法线向量沿 z 轴正向或者负向。对于拉伸体的每个侧表面，在表面上沿逆时针方向取三个顶点，计算外法线向量，并进行规范化处理。计算外法线向量和规范化处理的函数放在 OpenGL001View.h 文件下方或者 OpenGL001View.cpp 文件上方均可，代码如下：

```
void Unitlize(GLfloat * Vertex){     //向量规范化处理
GLfloat len = GLfloat(sqrt(Vertex[0] * Vertex[0] + Vertex[1] * Vertex[1] + Vertex[2] * Vertex
[2]));
    if(len == 0.0f)
        len = 1.0f;
    Vertex[0]/ = len;
    Vertex[1]/ = len;
    Vertex[2]/ = len;
}
void CaCulateNormal(GLfloat Vertices[3][3],GLfloat * Normal){     //计算法向量
    GLfloat v1[3],v2[3];
    //向量
    v1[0] = Vertices[1][0] - Vertices[0][0];
    v1[1] = Vertices[1][1] - Vertices[0][1];
    v1[2] = Vertices[1][2] - Vertices[0][2];
    v2[0] = Vertices[2][0] - Vertices[1][0];
    v2[1] = Vertices[2][1] - Vertices[1][1];
    v2[2] = Vertices[2][2] - Vertices[1][2];
    //向量叉乘
    Normal[0] = v1[1] * v2[2] - v1[2] * v2[1];
    Normal[1] = v1[0] * v2[2] - v1[2] * v2[0];
    Normal[2] = v1[2] * v2[0] - v1[0] * v2[2];
    //向量规范化处理
    Unitlize(Normal);
}
```

运行程序，绘制封闭的折线段，当形成多边形后进行点拉伸操作，即形成三维拉伸体，可进行平移、动画展示等操作。图 10.2-8 所示为一自定义的拉伸体的线框图和真实感显示效果，其中线框图中上下底面多边形已经网格化为多个三角形。

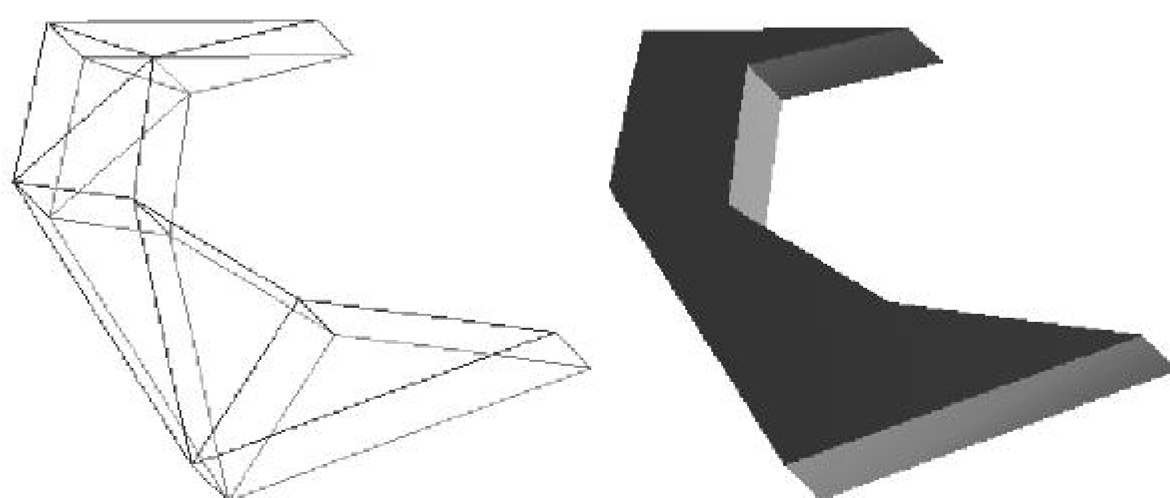


图 10.2-8 自定义拉伸体线框图及真实感显示

7. 颜色混合实现透明、反走样及雾化效果

在 OpenGL 中,通过将绘制图形的颜色混合可以实现透明、反走样等效果。所谓颜色混合是指按一定的规则将当前的绘图色(源颜色,C_s)与场景中已存在的对象的颜色(目标颜色,C_d)混合起来获得一种新的颜色。

使用混合功能,必须首先调用 glEnable()命令启用混合状态:glEnable(GL_BLEND),停止混合则调用 glDisable(GL_BLEND)命令。当混合功能启动后,源颜色与目标颜色混合的结果由混合方程控制。混合方程的一般形式为

$$C_t = C_s S + C_d D$$

其中,C_t 为混合后的颜色,S 和 D 分别为源混合因子和目标混合因子。设置混合因子需要通过调用 glBlendFunc()命令来实现。该命令原型为:

```
void glBlendFunc(GLenum sfactor,GLenum dfactor);
```

其中,参数 sfactor 和 dfactor 分别是源混合因子和目标混合因子对应的枚举常量。枚举常量的值以及对应各颜色分量 R、G、B 和 Alpha 的混合因子如表 10.2-5 所示。

表 10.2-5 混合因子

枚举常量	混合因子	意义
GL_ZERO	0	值为 0
GL_ONE	1	值为 1
GL_SRC_COLOR	(R_s, G_s, B_s, A_s)	值为源颜色值
GL_ONE_MINUS_SRC_COLOR	$(1-R_s, 1-G_s, 1-B_s, 1-A_s)$	值为 1－源颜色值
GL_DST_COLOR	(R_d, G_d, B_d, A_d)	值为目标颜色值
GL_ONE_MINUS_DST_COLOR	$(1-R_d, 1-G_d, 1-B_d, 1-A_d)$	值为 1－目标颜色值
GL_SRC_ALPHA	A_s	值为源颜色 Alpha 值
GL_ONE_MINUS_ALPHA	$1-A_s$	值为 1－源颜色 Alpha 值
GL_DST_ALPHA	A_d	值为目标颜色 Alpha 值
GL_ONE_MINUS_DST_ALPHA	$1-A_d$	值为 1－目标颜色 Alpha 值
GL_CONSTANT_COLOR	(R_c, G_c, B_c, A_c)	值为常量颜色值
GL_ONE_MINUS_CONSTANT_COLOR	$(1-R_c, 1-G_c, 1-B_c, 1-A_c)$	值为 1－常量颜色
GL_CONSTANT_ALPHA	A_c	值为常量 Alpha 值
GL_ONE_MINUS_CONSTANT_ALPHA	$1-A_c$	值为 1－常量 Alpha 值
GL_SRC_ALPHA_STAURE	$(F, F, F, 1)$	$F=\min\{A_s, 1-A_s\}$

例如，当混合因子设置命令为 glBlendFunc(GL_SRC_ALPHA，GL_ONE_MINUS_SRC_ ALPHA)时，则源颜色和目标颜色的混合因子分别为 A_s 和 $1-A_s$，然后 OpenGL 利用混合方程获得混合后的颜色值。

图 10.2-9 所示为颜色模型中常用的 RGB 模型，图中两种原色叠加产生的新颜色就是颜色混合的结果，其中的混合因子设置命令为 glBlendFunc(GL_ONE_MINUS_DST_COLOR，GL_ONE _MINUS _SRC _COLOR)。

图形的透明效果可以通过颜色混合实现，设置混合因子命令为 glBlendFunc (GL_SRC_ALPHA，GL_ONE_MINUS_SRC_ALPHA)。这时需要设置源颜色即后绘制图形的颜色 Alpha 值为大于 0 小于 1 的值。Alpha 值越小，透明度越高。

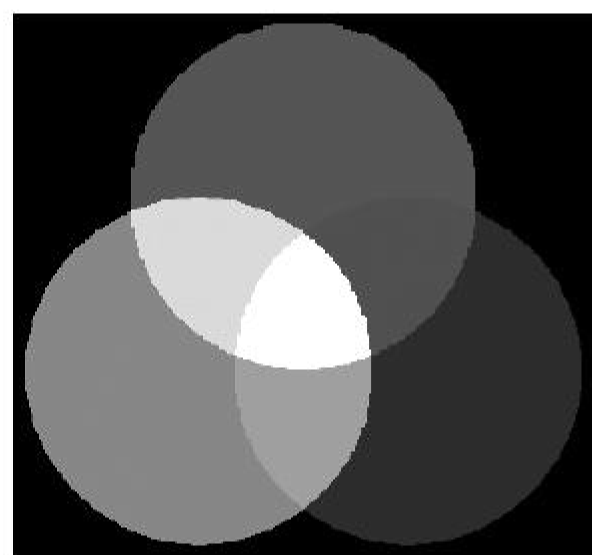


图 10.2-9 RGB 三基色混合

需要注意的是，当开启图形的深度检测功能后，颜色的混合和透明结果与图形的绘制顺序、图形在视线方向绘制的远近位置有关系。要实现混合，应该先绘制距离视线远的图形，再绘制距离视线近的图形，否则会出现不混合的现象。即使先绘制的距离视线近的图形的 Alpha 值为 0，也不会出现混合的情况，因为它已经绘出，在其后绘制的图形不能改变已绘制的图形的颜色。

在应用程序中实现颜色混合和透明时，首先，在视图类中建立一个变量标识：

```
int m_BlendFlag;
```

在视图类的构造函数中，设置 m_BlendFlag=0；然后，在菜单栏或者工具栏中，设置一个混合和一个透明的菜单 ID，并通过类向导建立对应的消息映射函数，并在消息函数中设置开启混合功能以及调用混合因子设置命令。例如，颜色混合的消息映射函数的代码为：

```
void COpenGL001View::OnBlend() {                          //混合
    if(m_BlendFlag == 0){
glBlendFunc(GL_ONE_MINUS_DST_COLOR,GL_ONE_MINUS_SRC_COLOR);
        glEnable(GL_BLEND);
        m_BlendFlag = 1;
    }
    else{
        glDisable(GL_BLEND);
        m_BlendFlag = 0;
    }
    Invalidate();
}
```

透明的消息映射函数的代码为：

```
void COpenGL001View::OnTransparent() {                    //透明
    if(m_BlendFlag == 0){
        glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
        glEnable(GL_BLEND);
        m_BlendFlag = 2;
    }
    else{
```

```
            glDisable(GL_BLEND);
            m_BlendFlag = 0;
        }
        Invalidate();
    }
```

在应用程序的绘图函数 RenderScene()中，增加混合和透明的相关绘图代码，以便显示混合的效果，代码放在函数的前面位置：

```
    if(m_BlendFlag!= 0){
        if(m_BlendFlag == 1){//混合
            GLUquadricObj * obj;                    //定义一个二次曲面指针
            glPushMatrix();
            glTranslatef(0.0,Win_Size/3,0.0);
            glColor4f(1.0,0.0,0.0,1.0);
            obj = gluNewQuadric();                  //生成二次曲面对象
            gluDisk(obj,0,Win_Size/3,30,1);
            glPopMatrix();
            glPushMatrix();
            glTranslatef( - Win_Size/5,0, - Win_Size/5);
            glColor4f(0.0,1.0,0.0,1.0);
            obj = gluNewQuadric();                  //生成二次曲面对象
            gluDisk(obj,0,Win_Size/3,30,1);
            glPopMatrix();
            glPushMatrix();
            glTranslatef(Win_Size/5,0, - Win_Size/3);
            glColor4f(0.0,0.0,1.0,1.0);
            obj = gluNewQuadric();                  //生成二次曲面对象
            gluDisk(obj,0,Win_Size/3,30,1);
            glPopMatrix();
        }
        else if(m_BlendFlag == 2){                  //透明
            glPushMatrix();
            glTranslatef(0.0,0.0, - Win_Size/2.0);
            glColor4f(1.0,0.0,0.0,1.0);
            m_3DRadius = Win_Size/4.0;
            if(m_bPolygonFill == TRUE)              //绘制一个圆环
                glutSolidTorus(m_3DRadius/2.0,m_3DRadius,30,30);
            else
                glutWireTorus(m_3DRadius/2.0,m_3DRadius,30,30);
            glPopMatrix();
            glPushMatrix();
            glTranslatef(Win_Size/4.0,Win_Size/4.0,Win_Size/2.0);
            glColor4f(0.0,1.0,0.0f,0.4f);
            glutSolidSphere(Win_Size/2.0,30.0,30.0f); //绘制一个球
            glPopMatrix();
        }
    }
```

注意，由于混合和透明与图形的绘制顺序、图形在视线方向绘制的远近位置有关系，因此，当开启深度检测或者改变图形绘制代码顺序时，上述代码产生的图形有可能不会出现混

合的效果。

直线的反走样也可以利用混合来实现。由于显示器的显示点为离散像素点，在绘制连续的直线时会产生锯齿那样的失真现象，反走样是指利用一定的方法减轻或者消除直线上的锯齿，使直线显得平整光滑。OpenGL 利用混合进行反走样，实际上是在锯齿的凹凸处填上介于直线颜色与背景颜色之间的颜色，从而在宏观上使直线显得比较光滑平顺。

直线反走样也要开启混合状态，混合因子设置命令的参数和图形透明设置的混合因子参数相同，也是：

```
glBlendFunc (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
```

然后，利用 glEnable(GL_LINE_SMOOTH)命令启用直线反走样功能，再调用 glHint()命令控制特定的渲染行为，该命令不仅用于控制直线的处理，也用于其他对象的处理。该命令是 OpenGL 中唯一一个与硬件有关的命令，若硬件不支持，则被忽略。命令原型为：

```
void glHint(GLenum target,GLenum mode);
```

其中，参数 target 可取的枚举常量值有：

GL_FOG_HINT——根据顶点或者根据每个像素计算雾；

GL_LINE_SMOOTH_HINE——设置直线采样质量；

GL_PERSPECTIVE_CORRECTION_HINT——颜色或者纹理贴图质量；

GL_POINT_SMOOTH_HINT——设置点采样质量；

GL_POLYGON_SMOOTH_HINT——多边形边界质量。

参数 mode 的取值有：

GL_FASTEST——高效率；

GL_NICEST——高质量；

GL_DONT_CARE——无特别要求。

在应用程序中实现反走样时，在工具栏或者菜单栏中建立一个 ID 标识，通过类向导建立其消息映射函数，在该函数中开启反走样。代码参考如下：

```
void COpenGL001View::OnLineSmooth() {                        //直线反走样
    if(m_BlendFlag == 0){
        glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);   //混合
        glEnable(GL_BLEND);
        glEnable(GL_LINE_SMOOTH);
        glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
        glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
        m_BlendFlag = 3;
    }
    else{
        glDisable(GL_BLEND);
        glDisable(GL_LINE_SMOOTH);
        m_BlendFlag = 0;
    }
    Invalidate();
}
```

颜色混合也可以营造雾的效果。为了表现雾效果，必须调用 glEnable(GL_FOG)启用雾状态，还需要调用 glFog()命令设置雾的参数。该命令原型为：

```
void glFogX(GLenum pname, Type param);
```

其中，X 为 f、i、fv 和 iv 之一。Type param 表示 GLfloat、GLint 两种数据类型的数组(向量)。pname 的枚举常量取值有：

GL_FOG_MODE——雾的模式，取值：GL_LINEAR(线形)、GL_EXP(指数)或者 GL_EXP2(平方指数)；

GL_FOG_DENSITY——雾的密度(单一值)；

GL_FOG_START——雾开始处到视点的距离(单一值)；

GL_FOG_END——雾结束处到视点的距离(单一值)；

GL_FOG_INDEX——雾颜色的索引值(单一值)；

GL_FOG_COLOR——雾的颜色，为包含 RGBA 的数组。

在应用程序中使用雾时，首先在视图类中建立雾的模式变量："int m_fogMode;"，并在视图类的构造函数中设置"m_fogMode=0;"，在菜单栏或者工具栏中设置雾的 ID 标识，通过类向导建立对应的消息映射函数，在消息映射函数中开启和设置雾状态，代码参考如下：

```
void COpenGL001View::OnFog() {
    if(m_fogMode == 0){                                  //雾的格式
        glEnable(GL_FOG);
        glFogi(GL_FOG_MODE, GL_LINEAR);                  //线性雾模式
        m_fogMode = 1;
        GLfloat fogColor[] = {0.5, 0.5f, 0.5f, 1.0};
        glFogfv(GL_FOG_COLOR, fogColor);
        glFogf(GL_FOG_DENSITY, 0.35F);
        glHint(GL_FOG_HINT, GL_DONT_CARE);
        glFogf(GL_FOG_START, - Win_Size);
        glFogf(GL_FOG_END, Win_Size);
    }
    else{
        glDisable(GL_FOG);
        m_fogMode = 0;
    }
    Invalidate();
}
```

如果采用其他雾的模式，则通过菜单栏或者工具栏设置其他雾模式即可。

8. OpenGL 图形选取交互技术

当鼠标在绘图窗口中移动时，单击某个图形，应用程序能够确定单击到的是场景中的哪个对象，这种技术称为图形的拾取，又称选择。图形拾取技术是实现图形的交互式操作非常重要的一个功能，OpenGL 对图形拾取也提供了相应的功能。

OpenGL 选择功能的原理是，以单击坐标的位置为中心创建一个比较小的可视区域，在选择模式下绘制图形，并使用 OpenGL 的选择特性测试可视区域内包含的图形对象，即为单击拾取的图形。

为了辨识拾取的图形对象，在绘制图形时，将图形进行命名，被拾取的图形名字被压入名字堆栈，以便判断。创建图形名字堆栈时，首先调用 glInitNames()命令创建并初始化名字堆栈为空。该命令原型为：

```
void glInitName();
```

为了保证名字堆栈中至少存在一条记录，创建名字堆栈后，立刻向其压入一条空记录，命令为 glPushName(0)。该命令的原型为：

```
void glPushName(GLuint name);
```

其中，参数 name 为无符号整数，该命令用于将参数所指的名字压入名字堆栈。

在绘制每一个图形前，首先调用 glLoadName()命令，将与图形对应的名字装入名字堆栈。该命令的原型为：

```
void glLoadName(GLuint name);
```

其中，参数 name 与 glPushName()的参数相同，用于向名字堆栈中装入一个名字，并用新名字替换栈顶的名字。

当多个图形对象之间存在一个隶属于另一个的层次关联关系时，例如身体的四肢隶属躯体，在拾取时，要求记录它们之间的这种层次关系。在绘制图形时，首先绘制所隶属的图形例如躯体，装入躯体名字到堆栈，再绘制隶属的图形例如四肢，将每个肢体的名字压入堆栈。在绘制每一个肢体时，肢体名字压入堆栈，绘制完该肢体后，还应将该肢体的名字再弹出堆栈，以保证在绘制过程中名字堆栈的栈顶仅保存一个名字，使下一个要绘制的图形名字出现在栈顶；否则，在拾取过程中，下一个图形对象的名字不可能出现。名字弹出堆栈的命令原型为：

```
void glPopName();
```

上述的名字堆栈操作命令，只在 OpenGL 的图形渲染模式为选择模式时才起作用，在其他模式下，这些命令将被忽略。

单击图形后，需要将单击记录保存在一个选择缓冲区中。建立选择缓冲区的命令为 glSelectBuffer()，该命令原型为：

```
voidglSelectBuffer(GLzisei size,GLuint * buffer);
```

其中，参数 size 为选择缓冲区的大小，参数 buffer 用来作为选择缓冲区的首地址。选择缓冲区实际上是一个无符号整形数组，每条单击记录在数组中至少占据四个元素：第一个元素包含单击事件发生时，名称堆栈中具有层次关系图形的数量；接下来两个元素包含上一条单击记录在可视窗口的最小和最大 z 坐标；第四个元素为名称堆栈的底部，即图形对象的名字；如果超过一个名字出现在名字堆栈中，它们将出现在该元素之后。

为了判断单击对象，调用 glRenderMode(GL_SELECT)命令将图形渲染模式设置为选择模式。图形渲染模式设置命令原型为：

```
void glRenderMode(GLenum mode);
```

其中，参数 mode 可为 GL_RENDER(默认绘图模式)、GL_SELECT(选择模式)或 GL_

FEEDBACK(反馈模式)。该命令将当前模式设置成参数所指定的模式,并返回上一次调用该命令时所设置的相应值。

在选择模式下,重新设置投影模式,建立拾取区域,对拾取区域创建和绘图方式相同的坐标投影矩阵,并重新绘制图形。拾取区域通过 gluPickMatrix()命令建立,该命令原型为:

```
void gluPickMatrix(GLdouble x,GLdouble y,GLdouble width,GLdouble height,GLint viewport[4]);
```

其中 x 和 y 为拾取区域的中心坐标,可以单击的窗口坐标点为中心坐标;width 和 height 为拾取区域以像素计的宽度和高度;参数 viewport 为当前的绘图窗口的尺寸数组,分别为窗口的左边界、上边界、右边界和下边界,可通过 glGetIntegerv(GL_VIEWPORT, viewport)命令获得,也可以通过 glViewportv()命令获得。

创建拾取区域后,再对拾取区域创建和绘图模式相同的坐标投影矩阵,然后绘制图形,绘制完成后再调用 glRenderMode(GL_RENDER)命令返回渲染模式,该命令同时返回在选择模式下单击的记录数,并将记录复制到选择缓冲区中。通过对缓冲区内容的进一步解析,即可判断出单击拾取的是哪一个图形对象。

为了在应用程序 OpenGL001 中实现图形拾取,首先在 OpenGL001View.h 头文件中建立选择的宏常量:

```
#define SELECT    21                              //选取
```

在 COpenGL001View 视图类中建立处理选择的函数:

```
void DoSelection(GLfloat xPos,GLfloat yPos);
```

然后在程序的工具栏或者菜单栏中设置一个选择的标识符,通过类向导创建消息映射函数,在该函数中设置 m_flag=SELECT,然后利用绘图函数 RenderScene()绘制可供选择的图形对象,参考代码如下:

```
if(m_flag == SELECT){
    glPushMatrix();
        glInitNames();                          //创建名字堆栈
        glPushName(0);                          //名字堆栈中压入 0
        glColor3f(1.0f,1.0f,0.0f);
        glPushMatrix();
            glTranslatef(-Win_Size/2.0,Win_Size/2.0,0.0f);
            glLoadName(SPHERE);                 //将球体的名字压入堆栈
            glutSolidSphere(Win_Size/4.0,30,30); //绘制球体
            glTranslatef(-Win_Size/4.0,-Win_Size/4.0,0.0f);
            glColor3f(0.0f,0.0f,0.5f);
            glPushName(CUBE);                   //和球有层次关系的立方体名字压入堆栈
            glutSolidCube(Win_Size/5.0);        //绘制立方体
            glPopName();                        //堆栈中弹出立方体名字
            glTranslatef(Win_Size/2.0,0.0,0.0f);
            glColor3f(0.0f,0.0f,0.5f);
            glPushName(SPHERE);                 //和球有层次关系的球体名字压入堆栈
            glutSolidSphere(Win_Size/5.0,30,30); //绘制球体
            glPopName();                        //堆栈中弹出球名字
        glPopMatrix();
```

```
        glColor3f(0.5f,0.0f,0.0f);
        glPushMatrix();
            glTranslatef(Win_Size/2.0,Win_Size/2.0,0.0f);
            glLoadName(CUBE);                   //将立方体的名字装入堆栈
            glutSolidCube(Win_Size/4.0);        //绘制立方体
        glPopMatrix();
        glColor3f(0.5f,0.5f,1.0f);
        glPushMatrix();
            glTranslatef(-Win_Size/2.0,-Win_Size/2.0,0.0f);
            glLoadName(CONE);                   //将圆锥体的名字压入堆栈
            glutSolidCone(Win_Size/4.0,Win_Size/4.0,20,20);    //绘制圆锥体
        glPopMatrix();
        glColor3f(0.0f,0.0f,1.0f);
        glPushMatrix();
            glTranslatef(Win_Size/2.0,-Win_Size/2.0,0.0f);
            glLoadName(TEAPOT);                 //将茶壶的名字压入堆栈
            glutSolidTeapot(Win_Size/4.0);      //绘制茶壶
        glPopMatrix();
    glPopMatrix();
}
```

当 m_flag=SELECT 时，在绘图窗口绘制可以选择的图形，这时，单击图形，在单击消息函数 OnLButtonDown()中设置调用处理图形选择的函数 DoSelection()：

```
void COpenGL001View::OnLButtonDown(UINT nFlags, CPoint point) {
    //原有代码此处省略
    if(m_flag == SELECT){                       //选取
        DoSelection(point.x,point.y);
    }
    CView::OnLButtonDown(nFlags, point);
}
```

处理选择图形的 DoSelection()函数代码如下：

```
void COpenGL001View::DoSelection(GLfloat xPos,GLfloat yPos){
    GLint hits,viewport[4];
    int cx,cy;
    GLuint SelBuff[64];
    //创建选择缓冲器
    glSelectBuffer(64,SelBuff);
    //调用矩阵操作函数,切换到投影矩阵模式,然后将其压栈,并置为单位矩阵
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    //生成拾取矩阵
    glGetIntegerv(GL_VIEWPORT,viewport);        //获得窗口数据
    gluPickMatrix(xPos,viewport[3]-yPos+viewport[1],2,2,viewport);    //生成拾取矩阵
    //设置与渲染模式下相同的透视投影矩阵
    cx = winWidth * 2;
    cy = winHeight * 2;
    glViewport(0, 0, cx, cy);
```

```
    if(cx<cy){
        winWidth = Win_Size;
        winHeight = Win_Size/aspect_ratio;
    }
    else{
        winWidth = Win_Size * aspect_ratio;
        winHeight = Win_Size;
    }
    glOrtho(-winWidth,winWidth,-winHeight,winHeight,-Win_Size*5,Win_Size*5);
    winWidth = cx/2.0;
    winHeight = cy/2.0;
    //切换渲染模式到 SELECT 模式
    glRenderMode(GL_SELECT);
    //渲染并搜集单击
    RenderScene();
    hits = glRenderMode(GL_RENDER);
    //返回投影模式
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    //返回模型绘图模式
    glMatrixMode(GL_MODELVIEW);
    //处理选择拾取对象
    if(hits == 1){
        switch(SelBuff[3]){
        case SPHERE:
            int count = SelBuff[0];
            if(count == 2){
                if(SelBuff[4] == CUBE)
                    AfxMessageBox("点中了球-立方体!");
                else if(SelBuff[4] == SPHERE)
                    AfxMessageBox("点中了球-球!");
                }
            else
                AfxMessageBox("点中了球!");
            break;
        case CUBE:
            AfxMessageBox("点中了立方体");
            break;
        case CONE:
            AfxMessageBox("点中了圆锥体");
            break;
        case TEAPOT:
            AfxMessageBox("点中了茶壶");
            break;
        }
    }
}
```

运行程序,单击设置的菜单栏或工具栏中的选择图标,绘制可供选择的图形,然后在图形上单击,即可弹出对话框指出选择的图形,效果如图 10.2-10 所示。



图 10.2-10　OpenGL 图形拾取功能

OpenGL 除了提供图形选择功能外，还提供对图形详细数据信息的反馈功能，反馈和选择的相似之处是分别在选择和反馈渲染模式下重新绘图，得到的信息都存在各自的缓冲器中，以便于应用程序使用。不过，选择缓冲器存储的只是拾取图形对象的部分信息，反馈缓冲器返回的是根据需求指定的所有图形对象的信息，例如，经过处理和变换后图元的类型——点、直线、多边形、图像、位图和自定义的标记，以及图元的顶点、颜色或者其他数据，返回的数据是经过光照处理和几何变换操作的。因此，反馈缓冲器要求的数组长度比较大，否则不能正确返回。关于反馈功能本书不再详述，读者如感兴趣，可参阅相关书籍。

10.3　OpenGL 图像处理技术

10.3.1　位图图像

除了绘制图形外，OpenGL 还具有一定的图像处理功能。而图像在 OpenGL 中也有非常重要的作用，其最常见的用途是应用于纹理映射。OpenGL 中的图像分为位图图像和像素图像两种类型。所谓位图图像指图像的每个像素数据仅以一个二进制位的形式存在，即 0 或 1，它们分别表示在屏幕上该像素点的关闭和打开状态，也称为二值图像。因此，每一个像素只有黑白两种颜色，当设置当前绘图颜色后，位图中被点亮即数据 1 的像素用当前绘图颜色绘出。

由于位图的数据和显示的像素点之间是一一对应的映射关系，因此位图的数据结构非常简单，可以用 0 或者 1 表示每个像素位，也可用常用的十六进制一次表示多个像素位，例如 0x37，就可以表示 8 个像素位的图像状态，分别是 00110111。

当给出一个位图的数据后，在绘制位图时需要进行两步操作：设置位图的绘制位置和绘制位图图像。指定绘图位置调用的命令为 glRasterPos()，命令原型为：

```
void glRasterPosnX(Args);
```

这是一个命令族，共包含24条命令，n可为2、3或4，X可为d、f、i、s以及带v的形式，Args则根据n和X的不同有不同的参数类型和个数。该命令指定位图显示时的左下角坐标值，该坐标与当前的坐标投影方式有关。

绘制位图图像调用的命令是glBitmap()，命令原型为：

```
void glBitmap(GLsizei width,GLsizei height,GLfloat xorig,GLfloat yorig,GLfloat xmove,GLfloat
ymove,const GLubyte * bitmap);
```

其中，参数width和height分别为位图图像的宽度和高度，它们必须与实际位图图像的对应参数完全相等；参数xorig和yorig为位图本身的原点；xmove和ymove为以像素计的图像相对绘制位置的偏移量；bitmap为指向该位图图像数据的指针。

在应用程序OpenGL001中绘制位图图像时，首先设置位图图像的数据。例如，下面是一个32×32位的斑马头部的位图图像数据数组，代码如下：

```
GLubyte Zebra[] = {0x00,0x00,0x00,0x00,
        0x37,0x20,0x00,0x00,
        0x13,0x60,0x00,0x00,
        0x10,0x60,0x00,0x00,
        0x14,0x60,0x00,0x00,
        0x06,0x18,0x00,0x00,
        0x07,0x10,0x00,0x00,
        0x07,0x40,0x00,0x00,
        0x03,0x60,0x00,0x00,
        0x03,0x70,0x00,0x00,
        0x01,0x70,0x00,0x3e,
        0x02,0x78,0x00,0x78,
        0x01,0x18,0x01,0x60,
        0x01,0x48,0x0a,0x08,
        0x00,0x62,0x14,0x00,
        0x00,0xf6,0x43,0xe0,
        0x00,0x3d,0xcb,0x60,
        0x00,0x19,0x9d,0x40,
        0x00,0x08,0x22,0xc0,
        0x00,0x00,0x09,0x80,
        0x00,0x02,0x40,0x80,
        0x00,0x01,0x1b,0x80,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x28,0x00,
        0x00,0x00,0x29,0x00,
        0x00,0x00,0x49,0x80,
        0x00,0x01,0x99,0x80,
        0x00,0x00,0x00,0x00
    };
```

上述的位图图像数据中，数据存储是从下向上进行的，也就是说，数组中第一行的4个十六进制数对应的是位图图像中最下方的一行，数组中最后一行则表示图像上方第一行的图形信息。后面将要介绍的文件扩展名为bmp的像素图像也是这种存储方式。

设置位图图像数据后，接下来的步骤和实现其他图形功能类似，在 OpenGL001View.h 中增加位图的宏定义：

```
#define BITMAP 22
```

并在工具栏或者菜单栏中设置一个绘制位图的标识符，通过类向导建立消息映射函数，在该函数中设置 m_flag=BITMAP 和格式化，然后，在绘图函数中增加位图的绘图代码，参考如下：

```
if(m_flag == BITMAP) {
    glRasterPos3f( - Win_Size/3.0, - Win_Size/3.0,0.0f);
    glBitmap(32,32,0.0f,0.0f,0.0f,0.0f,Zebra);
}
```

这样，单击应用程序的绘制位图的标识，视图窗口中就可显示一个位图斑马头部的图像。

10.3.2　像素图像

OpenGL 中的像素图像是指具有一定结构的、对图像中的每个像素有详细描述的二维图形，常见的 bmp、jpg、tga、png、tiff 等格式的图像均属于像素图像。其中的 bmp 图像虽然又称为位图图像，但在 OpenGL 中属于像素图像。

像素图像与位图图像相似，但是屏幕矩形区域中的每个像素并不是由 1 个位表示的。在像素图像中，每个像素一般都包含好几段数据，包含更多的信息。例如，图像的每个像素可以存储完整的颜色(R、G、B、A)。它们的存储结构根据其图像类型和所拥有的颜色数的不同有极大的变化。

在显示图像时，需要注意的是，OpenGL 本身不提供从文件读取像素和图像以及把像素和图像保存到文件的功能，只对满足绘制图像命令参数要求的数据集进行显示操作。OpenGL 显示(绘制)像素图像的命令是 glDrawPixesl()，该命令的原型是：

```
void glDrawPixels(GLsizei width,GLsizei height,GLenum format,GLenum type,const GLvoid * pixels);
```

其中，参数 width 和 height 分别为图像的宽度和高度；指针参数 pixels 为指向图像像素数据的指针；参数 type 为图像像素的数据类型，其取值为表 10.3-1 所示的枚举常量之一。

表 10.3-1　像素数据类型

枚举常量	意　　义
GL_UNSIGNED_BYTE	无符号的 8 位整数
GL_BYTE	有符号的 8 位整数
GL_BITMAP	每一位用一个无符号的 8 位整数表示
GL_UNSIGNED_SHORT	无符号的 16 位整数
GL_SHORT	有符号的 16 位整数
GL_UNSIGNED_INT	无符号的 32 位整数
GL_INT	有符号的 32 位整数
GL_FLOAT	单精度浮点数

参数 format 为所给图像的格式,取值为表 10.3-2 所示的枚举常量之一。

表 10.3-2 图像格式

枚举常量	意义
GL_COLOR_INDEX	使用颜色索引
GL_STENCIL_INDEX	每个像素包含一个单一的模板值
GL_DEPTH_COMPONENT	每个像素包含一个单一的深度值
GL_RGBA	颜色以红、绿、蓝、Alpha 顺序表示
GL_RED	每个像素包含一种单一的红色成分
GL_GREEN	每个像素包含一种单一的绿色成分
GL_BLUE	每个像素包含一种单一的蓝色成分
GL_ALPHA	每个像素包含一种单一的 Alpha 成分
GL_RGB	颜色以红、绿、蓝顺序表示
GL_LUMINANCE	每个像素包含一种单一的亮度成分
GL_LUMINANCE_ALPHA	每个像素包含亮度和 Alpha 成分
GL_BGR_EXT	颜色以蓝、绿、红顺序表示
BL_BGBA_EXT	颜色以蓝、绿、红、Alpha 顺序表示

在调用 glDrawPixels()命令时,首先必须对要显示的图像格式有详细的了解,例如图像的尺寸、图像的颜色模式、图像数据的类型以及像素数据的存放地址等;否则,将可能导致该命令执行后无图像显示,或者显示一片极为混乱的画面。

从图像文件中读取图像或者保存图像到图像文件,OpenGL 需要借助辅助库或者第三方库才能实现。OpenGL 辅助库 glaux 中存在一个加载 bmp 图像的命令 auxDIBImageLoad(),该命令的原型为:

```
AUX_RGBImageRec * APIENTRY auxDIBImageLoadA(LPCSTR fileSpec);
```

其中,参数 fileSpec 为带全路径的图片文件,其扩展名必须是 bmp,该命令用于载入一个 bmp 文件所描述的图像,载入成功返回一个指向 AUX_RGBImageRec 结构的指针,否则返回空指针。AUX_RGBImageRec 结构的定义为:

```
typedef struct_AUX_RGBImageRec{
GLint sizeX,sizeY;
Unsigned char * data;
} AUX_RGBImageRec;
```

在加载图像文件前,可以调用 glPixelStore()命令指定像素的存储模式,该命令的原型为:

```
void glPixelStoref/i(Type pname,Type param);
```

参数 pname 指定了像素数据在内存中的存储形式,其可用值为表 10.3-3 中所示的枚举常量之一。

表 10.3-3　像素数据的内存存储方式

枚举常量	意　义
GL_PACK_SWAP_BYTES	不压缩，为真时，所有多字节在存储到内存时进行字节交换
GL_PACK_LSB_FIRST	不压缩，为真时，位图最左端的像素存在第0位而不是第7位
GL_PACK_ROW_LENGTH	不压缩，设置图像的宽度，若为0，使用 width 参数
GL_PACK_SKIP_PIXELS	不压缩，设置图像中水平跳过的像素数
GL_PACK_SKIP_ROWS	不压缩，设置图像中垂直跳过的像素数
GL_PACK_ALIGNMENT	不压缩，设置图像中每条扫描线的对齐方式
GL_UNPACK_SWAP_BYTES	不压缩，为真时，所有多字节从内存中读取时进行字节交换
GL_UNPACK_LSB_FIRST	不压缩，为真时，位图最左端的像素在第0位而不是第7位
GL_UNPACK_ROW_LENGTH	不压缩，设置图像的宽度，若为0，使用 width 参数
GL_UNPACK_SKIP_PIXELS	不压缩，设置图像中水平跳过的像素数
GL_UNPACK_SKIP_ROWS	不压缩，设置图像中垂直跳过的像素数
GL_UNPACK_ALIGNMENT	不压缩，设置图像中每条扫描线的对齐方式

其中的 GL_UNPACK_ALIGNMENT 指图像像素存储时一条扫描线的对齐方式，该参数的取值只能为1、2、4或8，默认值是4，这正好与位图图像及像素图像 bmp 的图像一致。该值取1时，实际上将图像进行了压缩，即将一条扫描线中多余的空白字节删除掉。

在应用程序 OpenGL001 中显示像素图像时，首先在 OpenGL001View.h 中增加像素图像的宏定义：

```
#define IMAGE_FILE23
```

在视图类 COpenGL001View 中，增加图像数据的相关变量：

```
GLint m_iWidth,m_iHeight;                    //图像宽度和高度
GLubyte *m_pImage;                           //图像字节数据
```

其中，m_pImage 在视图的构造函数中，设置 m_pImage=NULL。

在工具栏或者菜单栏建立打开图像文件的 ID 标识，或者利用应用程序默认的打开文件标识 ID_FILE_OPEN，然后在视图里建立其消息映射函数，并在该函数中加载图像文件，加载图像文件可以借助于 MFC 的文件打开对话框来选择文件，并设置绘图标识。代码如下：

```
void COpenGL001View::OnFileOpen() {
//利用文件对话框打开图像文件
CFileDialog hFileDlg(true,NULL , NULL,OFN_FILEMUSTEXIST| OFN_READONLY | OFN_PATHMUSTEXIST,
TEXT("bmp 文件(*.bmp)|*.bmp|"), NULL);
    if(hFileDlg.DoModal() == IDOK) {
        AUX_RGBImageRec * m_image;
        glPixelStorei(GL_UNPACK_ALIGNMENT,1);             //设置像素存储格式
        m_image = auxDIBImageLoad(hFileDlg.GetPathName()); //加载图像
        m_iWidth = m_image->sizeX;                        //保存图像数据
        m_iHeight = m_image->sizeY;
        m_pImage = m_image->data;
        m_flag = IMAGE_FILE;                              //设置为绘制像素图像的标识
        InitOperation();
    }
}
```

在绘图函数 RenderScene()中增加显示像素图像的代码，如下：

```
if(m_flag == IMAGE_FILE){
    if(m_pImage)
glDrawPixels(m_iWidth,m_iHeight,GL_RGB,GL_UNSIGNED_BYTE,m_pImage);
}
```

执行程序，打开一个扩展名为 bmp 的图像文件，图像显示在绘图窗口。从显示的效果看，图像以窗口的左下角为起点，按像素绘制，当窗口比图像大时，图像在窗口能完全显示；当窗口比图像小时，图像右边和上边超出绘图窗口的部分将不能显示。

10.3.3 图像操作

OpenGL 也支持对显示的图像进行一些操作，例如，缩放、镜像、颜色分离、灰度图以及负像等，这些操作与计算机图像处理中的操作类似。

1. 图像缩放及反射

OpenGL 通过调用 glPixelZoom()命令实现图像的缩放，该命令原型为：

```
void glPixelZoom(GLfloat xfactor,GLfloat yfactor);
```

其中，参数 xfactor 和 yfactor 分别为水平方向和垂直方向的缩放系数。其值大于 1 时，图像放大；小于 1 但是大于 0 时，图像缩小。

设置图像缩放系数后，调用 glDrawPixels()命令显示图像。

当缩放系数小于 0 时，则图像不仅发生了缩放，而且还产生了反射，若图像在两个方向均发生反射变换，命令为 glPixelZoom(－1，－1)，则其原始的起点就由左下角变成了右上角。为了保证图像能被正确绘出，在绘制前必须将当前光栅位置也重新定位到绘图窗口右上角，调用命令为：

```
glRasterPos2i(m_Right,m_Top);
```

2. 颜色分离

利用颜色通道分离得到单一颜色通道的图像，这在计算机图像处理技术中有着极为重要的意义，例如可以分析某一颜色的分布及密度。对于 RGB 彩色图像，颜色通道有红色(R)通道、绿色(G)通道、蓝色(B)通道。颜色通道分离需要调用 glPixelTransfer()命令，其原型为：

```
void glPixelTransferf/i(Type pname,Type Type param);
```

其中，参数 pname 为要设置的像素转换参数名，取值为表 10.3-4 中的值；param 为参数 pname 的具体值。该命令用于将图像按参数所规定的像素属性进行迁移。

表 10.3-4 像素转换参数

参 数	类 型	默 认 值	取 值 范 围
GL_MAP_COLOR	GLboolean	GL_FALSE	GL_TRUE/GL_FALSE
GL_MAP_STENCIL	GLboolean	GL_FALSE	GL_TRUE/GL_FALSE
GL_INDEX_SHIFT	GLint	0	$(-\infty,\infty)$

续表

参　数	类　型	默 认 值	取 值 范 围
GL_INDEX_OFFSET	GLint	0	(−∞,∞)
GL_RED_SCALE	GLfloat	1.0	(−∞,∞)
GL_GREEN_SCALE	GLfloat	1.0	(−∞,∞)
GL_BLUE_SCALE	GLfloat	1.0	(−∞,∞)
GL_ALPHA_SCALE	GLfloat	1.0	(−∞,∞)
GL_DEPTH_SCALE	GLfloat	1.0	(−∞,∞)
GL_RED_BIAS	GLfloat	0.0	(−∞,∞)
GL_GREEN_BIAS	GLfloat	0.0	(−∞,∞)
GL_BLUE_BIAS	GLfloat	0.0	(−∞,∞)
GL_ALPHA_BIAS	GLfloat	0.0	(−∞,∞)
GL_DEPTH_BIAS	GLfloat	0.0	(−∞,∞)

例如，当进行红色通道分离时，通过三次 glPixelTransfer()命令调用，将参数 GL_RED_SCALE 设置为 1，将 GL_GREEN_SCALE 和 GL_BLUE_SCALE 设置为 0，其意义是将图像中红色成分缩放系数设为 1，其他两种颜色成分缩放系数转为 0。最终效果是完全保留了图像中红色分量的值，而完全抛弃了绿色和蓝色分量的值。经过这样变换的图像，仅保留了原先的红色成分，再无其他颜色成分，从而达到将红色颜色通道分离出来的效果。

3. 灰度图

灰度图是一种无颜色的、仅用灰度级别来表示图像层次的图像，在计算机数字图像处理中有极为重要的作用。进行灰度变换时，图像将失去颜色信息，仅保留亮度信息。对于 bmp 图像而言，亮度信息隐式地存储在颜色信息中。

在进行灰度转换前，首先需要调用 glDrawPixels()命令将原图像绘制到颜色缓冲器中，以便于程序对其像素信息进行读取；接着，程序再分配一块与原图像大小完全相同的内存，以便存储转换后的像素信息；然后，调用 glPixelTransfer()命令对原图像进行像素迁移。根据美国国家电视标准委员会的标准，RGB 向亮度空间的转换公式为

$$\text{Luminance} = 0.3R + 0.59G + 0.11B$$

因此，需要调用 3 次 glPixelTransfer()命令，分别对红色、绿色和蓝色进行不同的缩放：

```
glPixelTransferf(GL_RED_SCALE,0.3f);
glPixelTransferf(GL_GREEN_SCALE,0.59f);
glPixelTransferf(GL_BLUE_SCALE,0.11f);
```

经过以上的准备工作，调用 glReadPixels()将颜色缓冲器中的像素信息读入系统分配的内存中。该命令原型为：

```
void glReadPixels(GLint x,GLint y,GLsizei width,GLsizei height,GLenum format, GLenum type,
GLvoid *pixels);
```

其中，参数 x 和 y 为图像的左下角坐标；参数 width 和 height 为图像的宽度和高度；参数 format 为图像的格式，为表 10.3-2 所示的枚举常量，因为此处分析的是亮度，所以 format 取值为 GL_LUMINANCE；参数 type 为像素的数据类型，为表 10.3-2 中的枚举常量；参数 pixels 为分配的与原图像大小完全相同的内存，存储读取的像素信息。

然后，再调用 glDrawPixels()命令(其中的参数 format 取值为 GL_LUMINANCE)绘制已经进行灰度处理分配在内存中的图像，即为灰度图。

最后，还需释放分配的内存，以防止内存泄漏。

4. 负像

负像如同摄影中的胶片，其颜色与最终的正片互为补色。从色彩学原理可知，补色的颜色值与原色的颜色值之和为常量。对于 RGB 三原色模型，设其各颜色值的取值范围为 0～1，则红、绿、蓝三个颜色的补色 C_R、C_G、C_B 为

$$\begin{cases} C_R = 1 - R \\ C_G = 1 - G \\ C_B = 1 - B \end{cases}$$

在代码中计算补色时，首先对一个长度为 256(R、G、B 颜色分量的灰度级别均为 256)的数组的每个元素按照上述补色理论进行赋值：

```
GLfloat invert[256];
Invert[0] = 0;
For(int i = 1;i < 256;i++)
    invert[i] = 1.0 - (GLfloat)i/255.0;
```

然后，通过调用 glPixelMap()命名对数组设置像素转换映射。该命令原型为：

```
void glPixelMapf/i/sv(GLenum map,GLsizei mapsize,const Type * values);
```

其中，参数 map 为像素转换映射名，其取值为表 10.3-5 中所示的枚举常量；参数 mapsize 为被定义为映射的大小，即数组大小；参数 values 为指向要定义映射的数组指针。

表 10.3-5 像素转换映射名

枚举常量	意义
GL_PIXEL_MAP_I_TO_I	将颜色索引映射成颜色索引
GL_PIXEL_MAP_S_TO_S	将模板索引映射成模板索引
GL_PIXEL_MAP_I_TO_R	将颜色索引映射成红色成分
GL_PIXEL_MAP_I_TO_G	将颜色索引映射成绿色成分
GL_PIXEL_MAP_I_TO_B	将颜色索引映射成蓝色成分
GL_PIXEL_MAP_I_TO_A	将颜色索引映射成 Alpha 成分
GL_PIXEL_MAP_R_TO_R	将红色成分映射成红色成分
GL_PIXEL_MAP_G_TO_G	将绿色成分映射成绿色成分
GL_PIXEL_MAP_B_TO_B	将蓝色成分映射成蓝色成分
GL_PIXEL_MAP_A_TO_A	将 Alpha 成分映射成 Alpha 成分

将 R、G、B 颜色按补色绘制时，首先调用三次 glPixelMap()命令：

```
glPixelMapfv(GL_PIXEL_MAP_R_TO_R,255,invert);
glPixelMapfv(GL_PIXEL_MAP_G_TO_G,255,invert);
glPixelMapfv(GL_PIXEL_MAP_B_TO_B,255,invert);
```

然后，再调用 glPixelTransferi(GL_MAP_COLOR,GL_TRUE)命令启用像素映射。

这样，在执行命令 glDrawPixel()时，各颜色按其补色绘制。例如，原图像某一像素

RGB 值为(0.4,0.3,0.7),调用上述三次像素转换映射命令,绘制为(0.6,0.7,0.3)。

5. 图像操作实例

程序 OpenGL001 中实现图像操作时,首先在视图类头文件增加相关图像操作的宏定义:

```
#define IMAGEZOOM          24
#define IMAGEREVERSE       25
#define IMAGERED           26
#define IMAGEGREEN         27
#define IMAGEBLUE          28
#define IMAGEGRAYMAP       29
#define IMAGEINVERSE       30
```

在视图类中建立一个图像操作的变量,例如 int flag_image,并在打开图像文件的函数中增加其初始值设置,如 flag_image=0。

然后,在工具栏或者菜单栏建立相关图像操作的 ID 标识符,并通过类向导建立对应的消息映射函数,在该消息映射函数中,将对应的图像操作宏赋给图像操作变量。例如,图像缩放操作消息映射函数的代码为:

```
void COpenGL001View::OnImageZoom() {
    if(m_flag == IMAGE_FILE){
        flag_image = IMAGEZOOM;
        Invalidate();
    }
}
```

为每个图像操作消息映射函数设置对应的宏后,在绘图函数 RenderScene()中统一绘制图像,将前文中对应的像素图像绘制代码修改如下:

```
if(m_flag == IMAGE_FILE){
    if(m_pImage){
        GLbyte * pModefied = NULL;
        GLint i;
        switch(flag_image){
        case IMAGEZOOM:
            glPixelZoom(0.5f,0.5f);
            break;
        case IMAGEREVERSE:
            glPixelZoom(-1.0f,-1.0f);
            if(aspect_ratio<1)
                glRasterPos2i(Win_Size,Win_Size/aspect_ratio);
            else
                glRasterPos2i(Win_Size * aspect_ratio,Win_Size);
            break;
        case IMAGERED:
            glPixelTransferf(GL_RED_SCALE,1.0f);
            glPixelTransferf(GL_GREEN_SCALE,0.0f);
            glPixelTransferf(GL_BLUE_SCALE,0.0f);
            break;
```

```
        case IMAGEGREEN:
            glPixelTransferf(GL_RED_SCALE,0.0f);
            glPixelTransferf(GL_GREEN_SCALE,1.0f);
            glPixelTransferf(GL_BLUE_SCALE,0.0f);
            break;
        case IMAGEBLUE:
            glPixelTransferf(GL_RED_SCALE,0.0f);
            glPixelTransferf(GL_GREEN_SCALE,0.0f);
            glPixelTransferf(GL_BLUE_SCALE,1.0f);
            break;
        case IMAGEGRAYMAP:
glDrawPixels(m_iWidth,m_iHeight,GL_RGB,GL_UNSIGNED_BYTE,m_pImage);
            pModefied = (GLbyte * )new GLbyte[m_iWidth * m_iHeight];
            glPixelTransferf(GL_RED_SCALE,0.3f);
            glPixelTransferf(GL_GREEN_SCALE,0.59f);
            glPixelTransferf(GL_BLUE_SCALE,0.11f);
glReadPixels(0,0,m_iWidth,m_iHeight,GL_LUMINANCE,GL_UNSIGNED_BYTE,pModefied);
            glPixelTransferf(GL_RED_SCALE,1.0f);
            glPixelTransferf(GL_GREEN_SCALE,1.0f);
            glPixelTransferf(GL_BLUE_SCALE,1.0f);
            break;
        case IMAGEINVERSE:
            GLfloat invert[256];
            invert[0] = 0;
            for(i = 1;i < 256;i++)
                invert[i] = 1.0 - i/255.0;
            glPixelMapfv(GL_PIXEL_MAP_R_TO_R,255,invert);
            glPixelMapfv(GL_PIXEL_MAP_G_TO_G,255,invert);
            glPixelMapfv(GL_PIXEL_MAP_B_TO_B,255,invert);
            glPixelTransferi(GL_MAP_COLOR,GL_TRUE);
            break;
        default:
            break;
        }
        if(pModefied == NULL)
glDrawPixels(m_iWidth,m_iHeight,GL_RGB,GL_UNSIGNED_BYTE,m_pImage);
        else{
glDrawPixels(m_iWidth,m_iHeight,GL_LUMINANCE,GL_UNSIGNED_BYTE,pModefied);
            delete pModefied;
        }
        //为不影响以后的绘图操作,再调用图像操作命令恢复现场
        glPixelTransferi(GL_MAP_COLOR,GL_FALSE);
        glPixelTransferi(GL_RED_SCALE,1.0F);
        glPixelTransferi(GL_GREEN_SCALE,1.0F);
        glPixelTransferi(GL_BLUE_SCALE,1.0F);
        glPixelZoom(1.0f,1.0f);
        if(aspect_ratio < 1)
            glRasterPos2i( - Win_Size, - Win_Size/aspect_ratio);
        else
            glRasterPos2i( - Win_Size * aspect_ratio, - Win_Size);
    }
}
```

运行上述程序,打开图像文件即可实现对图像的缩放、反射、颜色分离、绘制灰度图以及负像等效果。

10.4 OpenGL 纹理映射技术

10.4.1 纹理映射的一般步骤

纹理映射又称纹理贴图,是将一幅图像粘贴在二维或三维图形对象上,使得对象表面不再是简单的纯色或者过渡色,而表现为具有一定材质、纹理或图案的表面细节结构。从本质上讲,纹理是一个数组,其中的数据为颜色、灰度或颜色加 Alpha 值,纹理数组的值通常被称为纹素。OpenGL 使用纹理映射时,一般需要执行以下步骤。

第一步,创建纹理对象并为其指定纹理。

大部分情况下,纹理是二维的,如同图像一样,也可以是一维或者三维的。在描述纹理的数据中,每个纹素可以使用 1～4 个数据来表示(R、G、B、A)四元组、调整常量和深度常量。

第二步,确定纹理如何应用到每个片元的像素上。

可以使用四种方法根据片元和纹理图像数据来计算最终的 RGBA 值。第一种方法就是简单地使用纹理颜色作为最终片元的颜色;第二种方法是用纹理替换片元上的图案和颜色,这种方式又称为替换(replace)模式,如同贴画一样,将纹理绘制到片元上;第三种方法是使用纹理来调整片元的颜色,这种方法对于组合光照和纹理效果非常有用;第四种方法是根据纹理值,用一种常量颜色与片元的颜色进行混合。

第三步,启用纹理映射。

在进行纹理映射前,必须将 GL_TEXTURE_1D、GL_TEXTURE_2D、GL_TEXT URE_3D 或 GL_TEXTURE_CUBE_MAP 作为参数,调用 glEnable()命令启用相应维数的纹理映射功能,这些参数分别表示一维、二维、三维或者立方图纹理。如果同时启用了二维和三维纹理,则以后者为准;如果启用了 GL_TEXTURE_CUBE_MAP,则其他被启用的纹理均不再有效。为了使程序更清晰,应该只启用一种类型的纹理。

当不再使用纹理映射功能时,应当调用 glDisable()禁用相应的纹理映射。

第四步,绘制场景、提供纹理和几何坐标。

确定纹理在粘贴之前如何根据应用的片元进行排列,也就是说,在场景中指定物体后,需要同时提供纹理坐标和几何物体坐标,指定纹理坐标和几何物体对象之间的对应关系。对于二维纹理图,纹理坐标的范围都是 0.0～1.0,但是纹理贴图的物体坐标没有限制,因此,需要制定纹理坐标和物体坐标之间的对应关系。

在进行纹理映射前,读入纹理图像数据后,OpenGL 必须对纹理图形进行加载,其中,加载一维、二维和三维图像的命令不同。对应的函数原型分别如下:

```
void glTexImage1D(GLenum target,GLint level,GLint internalformat,GLsizei width,GLint border,
GLenum format,GLenum type,const GLvoid * pixels);
void glTexImage2D ( GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei
```

```
height,GLint border,GLenum format,GLenum type,const GLvoid * pixels);
void glTexImage3D (GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei
height,GLsizei depth,GLint border,GLenum format,GLenum type,const GLvoid * pixels);
```

其中,target 为纹理对象,其取值分别为 GL_TEXTURE_1D、GL_TEXTURE_2D 和 GL_TEXTURE_3D;参数 level 为纹理的 LOD(level of detail,层细节)值,仅用于 mipmap 贴图,一般情况下,该参数取值为 0;参数 internalformat 用于指定纹理图像的颜色成分,取值取决于图像本身,常用的纹理内部格式有:

GL_ALPHA——按照 Alpha 值存储纹理单元;

GL_LUMINANCE——按照亮度值存储纹理单元;

GL_LUMINANCE_ALPHA——按照亮度值和 Alpha 值存储纹理单元;

GL_RGB——按照红、绿、蓝成分存储纹理单元;

GL_RGBA——按照红、绿、蓝和 Alpha 成分存储纹理单元。

参数 width、height 和 depth 分别为纹理图像的宽度、高度和深度,应当特别指出的是,这几个值可以不同,但是必须是 2 的整数次幂,否则将导致映射失败;参数 border 为贴图时边框的宽度,其取值仅能为 0 或 1;参数 format 为纹理图像的格式,其取值为表 10.3-2 中的枚举常量;type 为像素的数据类型,取值为表 10.3-1 中的枚举常量;参数 pixels 为指向图像像素数据的指针。

纹理映射时,需要调用 glTexParameter()和 glTexEnv()命令为纹理映射设置参数和纹理环境,它们对纹理映射的效果影响极大。glTexParameter()命令的原型是:

```
void glTexParameterX(GLenum target,GLenum pname,Type param);
```

其中,X 可以是 f、i、fv 或 iv;参数 target 为纹理对象,其取值可为 GL_TEXTURE_1D、GL_TEXTURE_2D 和 GL_TEXTURE_3D;参数 pname 则用来指定纹理的过滤方式,其值为表 10.4-1 中的枚举常量。

表 10.4-1 参数 pname 的取值

枚举常量	意义
GL_TEXTURE_MIN_FILTER	返回纹理缩小过滤器值
GL_TEXTURE_MAG_FILTER	返回纹理放大过滤器值
GL_TEXTURE_MIN_LOD	返回纹理细节值的最小层
GL_TEXTURE_MAX_LOD	返回纹理细节值的最大层
GL_TEXTURE_BASE_LEVEL	返回最大 Mip 贴图值的基层
GL_TEXTURE_MAX_LEVEL	返回最大 Mip 贴图值的数量层
GL_TEXTURE_LOD_BIAS	纹理细节层偏向
GL_TEXTURE_WRAP_S	返回 S 坐标方向的环境模式
GL_TEXTURE_WRAP_T	返回 T 坐标方向的环境模式
GL_TEXTURE_WRAP_R	返回 R 坐标方向的环境模式
GL_TEXTURE_BORDER_COLOR	返回纹理边界颜色
GL_TEXTURE_PRIORITY	返回当前纹理优先设置
GL_TEXTURE_REIDENT	若纹理为常驻纹理,则返回 GL_TRUE
GL_DEPTH_TEXTURE_MODE	返回深度纹理模式

续表

枚举常量	意　　义
GL_TEXTURE_COMPARE_MODE	返回纹理比较模式
GL_TEXTURE_COMPARE_FUNC	返回深度纹理函数
GL_TEXTURE_GENERATE_MIPMAP	若自动 Mip 被启用，则返回 GL_TRUE

参数 param 为所指定的过滤器取值，可以是一个单一的实数或整数，也可以是一个数组。对于缩小纹理，该参数提供了 6 个过滤函数，分别由表 10.4-2 中的常量指定。

表 10.4-2　过滤函数

函　　数	意　　义
GL_NEAREST	在 Mip 基层上执行最邻近过滤
GL_LINEAR	在 Mip 基层上执行线性过滤
GL_NEAREST_MIPMAP_NEAREST	选择最邻近 Mip 层，并执行最邻近过滤
GL_LINEAR_MIPMAP_NEAREST	在 Mip 层之间执行线性插补，并执行最邻近过滤
GL_NEAREST_MIPMAP_LINEAR	选择最邻近 Mip 层，并执行线性过滤
GL_LINEAR_MIPMAP_LINEAR	在 Mip 层之间执行线性插补，并执行线性过滤

纹理环境是指如何组织纹理图像的颜色与几何对象的颜色。设置纹理环境需要调用 glTexEnv()命令，该命令的原型为：

```
void glTexEnvX(GLenum target,GLenum pname,Type param);
```

其中，X 的含义和 glTexParameter()命令原型中的相同；参数 target 也为纹理对象，取值可为 GL_TEXTURE_ENV 或 GL_TEXTURE_FILTER_CONTROL 等；参数 pname 为环境参数，对于非数组参数命令，其取值只能为 GL_TEXTURE_ENV_MODE，对于数组参数命令，还可以取 GL_TEXTURE_ENV_COLOR；参数 param 为所设置的环境参数值，可以为存放在数组中的 RGBA 值，也可以为具有一定意义的以下值：GL_MODULATE（调节）、GL_DECAL（贴纸）、GL_BLEND（混合）、GL_REPLACE（替换）。

OpenGL 在加载纹理图像、设置纹理映射参数和纹理环境，并启用纹理功能后，还需要把设置好的纹理图像和几何物体的具体位置进行对应，即建立纹理坐标点和图形坐标点的映射关系。纹理坐标的坐标系被命名为 s、t、r 和 q（类似于顶点坐标的 x、y、z 和 w），坐标值指定为浮点型，坐标利用 glTexCoord()函数指定，对应的一维、二维和三维纹理坐标函数原型分别为：

```
void glTexCoord1f(GLfloat s);
void glTexCoord2f(GLfloat s,GLfloat t);
void glTexCoord3f(GLfloat s,GLfloat t,GLfloat r);
```

无论纹理图像有多大，坐标值范围都是[0.0,1.0]。

在某一几何表面进行纹理映射时，在指定该表面上一个顶点的同时，必须给出纹理图像在该点的对应坐标值。例如，指定一个表面顶点及对应纹理坐标的代码如下：

```
glTexCoord2f(0.0f,0.0f);            //指定该点对应的纹理坐标
glVertex3f(-3.0,-5.0,0.0);          //指定该点坐标
```

根据上述步骤即可在程序中实现纹理映射，例如，在程序 OpenGL001 中实现纹理图像映射，首先，在视图类文件中定义一个纹理映射的宏：

```
#define TEXTURE_MAP_2D     31
```

在菜单栏或工具栏中创建一个装载纹理图像的图标标识，通过类向导在视图类中创建该标识的消息映射函数，例如为 OnImageMapLoad()，并在该函数中加载图像文件和设置纹理映射参数等，代码参考如下：

```
void COpenGL001View::OnImageMapLoad() {
    //加载图像文件
    CFileDialog hFileDlg(true, NULL , NULL, OFN_FILEMUSTEXIST | OFN_ READONLY | OFN_PATHMUSTEXIST ,
TEXT("bmp 文件( * .bmp)| * .bmp|"), NULL);
    if(hFileDlg.DoModal() == IDOK) {
        AUX_RGBImageRec * m_image;
        glPixelStorei(GL_UNPACK_ALIGNMENT,1);
        m_image = auxDIBImageLoad(hFileDlg.GetPathName());
        m_iWidth = m_image - > sizeX;
        m_iHeight = m_image - > sizeY;
        m_pImage = m_image - > data;
        //加载纹理图像
        glTexImage2D(GL_TEXTURE_2D,0,3,m_iWidth,m_iHeight,0,GL_RGB,GL_UNSIGNED_BYTE,
m_pImage);
        //设置纹理参数和环境参数
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER,GL_LINEAR);glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_
LINEAR);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);
        m_flag = TEXTURE_MAP_2D;          //设置纹理映射标识
        InitOperation();                  //初始化环境
        glEnable(GL_TEXTURE_2D);          //启用二维纹理映射
    }
}
```

在绘图函数 RenderScene()中加入绘制纹理的代码，此代码放在最后一行代码 glPopMatrix()之前即可，参考如下：

```
if(m_flag == TEXTURE_MAP_2D){
        glBegin(GL_QUADS);
            glTexCoord2f(0.0f,0.0f);
            glVertex3f( - Win_Size/3.0, - Win_Size/3.0,0.0);
            glTexCoord2f(1.0,0.0);
            glVertex3f(Win_Size/3.0, - Win_Size/3.0,0.0);
            glTexCoord2f(1.0,1.0);
            glVertex3f(Win_Size/3.0,Win_Size/3.0,0.0);
            glTexCoord2f(0.0,1.0);
            glVertex3f( - Win_Size/3.0,Win_Size/3.0,0.0);
        glEnd();
    }
```

这样，执行应用程序，加载一个图像文件后，纹理图像就映射在一个四边形表面上，该纹理图像可以随着四边形一起进行几何变换。

纹理的图像除了可以是像素图像外，也可以是位图图像，位图图像的纹理加载方式和像素图像的加载方式完全相同，只是图像的格式和尺寸大小为位图本身的格式和尺寸大小。

需要注意的是，设置不同的纹理设置的参数和纹理映射的环境参数对纹理效果有很大影响，可以通过设置不同的参数进行比较，此处不再赘述。

10.4.2　纹理对象

当物体有多个表面都要进行纹理贴图时，多个表面可以贴同一个纹理图像，也可以分别贴同一个纹理图像的不同部分。当多个表面分别贴同一个纹理图像的不同部分时，需要将纹理图像的坐标进行分段，每一段分别和一个表面的坐标对应，以此实现同一个纹理图像的不同部分映射在不同的表面上。

物体的多个表面也可以分别映射不同的纹理图像，这时存在多个纹理对象，对于多个纹理图像存在纹理对象的管理调度问题。

为了产生多个纹理对象，首先，必须通过调用 glGenTextures()命令为每个纹理对象创建 ID 标识，该命令原型为：

```
void glGenTextures(GLsizei n,GLuint * textures);
```

其中，参数 n 为纹理对象的个数；textures 为存放产生的纹理对象 ID 标识的指针地址，当纹理对象是多个时，textures 以纹理对象数组变量的形式存放。

纹理对象 ID 产生后，必须调用 glBindTexture()命令将纹理对象和 ID 进行绑定，该命令原型为：

```
void glBindTexture(GLenum target,GLuint texture);
```

其中，参数 target 为纹理对象，取值为枚举常量：GL_TEXTURE_1D、GL_TEXTURE_2D 或者 GL_TEXTURE_3D；参数 texture 为对应的纹理对象 ID。

glBindTexture()命令的作用除了绑定纹理对象 ID 外，同时还设置该纹理对象为当前纹理，这时，可以对其进行纹理图像加载以及纹理参数和环境参数设置，设置方法和单个纹理的设置方法相同。另外，对于多个纹理对象，当要使用某一个纹理进行映射时，也必须首先调用 glBindTexture()命令将其设置为当前纹理。

在程序 OpenGL001 中举例实现多个纹理对象时，可以先在视图类文件中定义一个纹理对象的宏：

```
#define IMAGE_MAP_OBJECT    32
```

在视图类中，定义纹理对象的数组变量，假如纹理对象个数为 5，则纹理对象数组为：

```
GLuint ImageIDs[5];
```

然后，在应用程序工具栏或者菜单栏中设置一个纹理对象的图标标识，通过类向导在视图类中创建该图标标识的消息映射函数，例如为 OnImageMapObject()，在该函数中产生纹

理对象 ID,并进行绑定,以及加载纹理图像和设置纹理参数等。代码参考如下:

```
void COpenGL001View::OnImageMapObject() {
    glGenTextures(5,ImageIDs);              //产生纹理对象 ID
    for(int i=0;i<5;i++){                   //加入图像文件
CFileDialog hFileDlg(true,NULL,NULL,OFN_FILEMUSTEXIST | OFN_READONLY | OFN_PATHMUSTEXIST,
TEXT("bmp 文件(*.bmp)|*.bmp|"), NULL);
        if(hFileDlg.DoModal() == IDOK) {
            glBindTexture(GL_TEXTURE_2D,ImageIDs[i]);
            AUX_RGBImageRec * m_image;
            glPixelStorei(GL_UNPACK_ALIGNMENT,1);
            m_image=auxDIBImageLoad(hFileDlg.GetPathName());
            m_iWidth=m_image->sizeX;
            m_iHeight=m_image->sizeY;
            m_pImage=m_image->data;
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glTexImage2D(GL_TEXTURE_2D,0,3,m_iWidth,m_iHeight,0,GL_RGB,GL_UNSIGNED_BYTE,m_pImage);
         }
        else {
            CString str;
            str.Format("请选择第%d个映射图像!",i+1);
            AfxMessageBox(str);
            i--;
        }
    }
    m_flag=IMAGE_MAP_OBJECT;
    InitOperation();
    glEnable(GL_TEXTURE_2D);
}
```

在绘图函数 RenderScene()中加入映射多个纹理对象的代码,此代码放在最后一行代码 glPopMatrix()之前。在本实例中,将加载的 5 个纹理对象图像分别绑定映射到一个正方体的 6 个表面上,其中最后两个表面采用同一个纹理图像的不同坐标部分映射。代码参考如下:

```
if(m_flag==IMAGE_MAP_OBJECT){
        glBindTexture(GL_TEXTURE_2D,ImageIDs[0]);       //绑定到前表面
        glBegin(GL_POLYGON);
            glNormal3f(0.0,0.0,1.0);
            glTexCoord2f(0.0f,0.0f);
            glVertex3f(-Win_Size/3.0,-Win_Size/3.0,Win_Size/3.0);
            glTexCoord2f(1.0,0.0);
            glVertex3f(Win_Size/3.0,-Win_Size/3.0,Win_Size/3.0);
            glTexCoord2f(1.0,1.0);
            glVertex3f(Win_Size/3.0,Win_Size/3.0,Win_Size/3.0);
            glTexCoord2f(0.0,1.0);
```

```
    glVertex3f( - Win_Size/3.0,Win_Size/3.0,Win_Size/3.0);
glEnd();
glBindTexture(GL_TEXTURE_2D,ImageIDs[1]);       //绑定到后表面
glBegin(GL_POLYGON);
    glNormal3f(0.0,0.0, - 1.0);
    glTexCoord2f(0.0f,0.0f);
    glVertex3f( - Win_Size/3.0, - Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(0.0,1.0);
    glVertex3f( - Win_Size/3.0,Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(1.0,1.0);
    glVertex3f(Win_Size/3.0,Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(1.0,0.0);
    glVertex3f(Win_Size/3.0, - Win_Size/3.0, - Win_Size/3.0);
glEnd();
glBindTexture(GL_TEXTURE_2D,ImageIDs[2]);       //绑定到左表面
glBegin(GL_POLYGON);
    glNormal3f(1.0,0.0,0.0);
    glTexCoord2f(0.0f,0.0f);
    glVertex3f(Win_Size/3.0, - Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(0.0,1.0);
    glVertex3f(Win_Size/3.0,Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(1.0,1.0);
    glVertex3f(Win_Size/3.0,Win_Size/3.0,Win_Size/3.0);
    glTexCoord2f(1.0,0.0);
    glVertex3f(Win_Size/3.0, - Win_Size/3.0,Win_Size/3.0);
glEnd();
glBindTexture(GL_TEXTURE_2D,ImageIDs[3]);       //绑定到右表面
glBegin(GL_POLYGON);
    glNormal3f( - 1.0,0.0,0.0);
    glTexCoord2f(0.0f,0.0f);
    glVertex3f( - Win_Size/3.0, - Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(1.0,0.0);
    glVertex3f( - Win_Size/3.0, - Win_Size/3.0,Win_Size/3.0);
    glTexCoord2f(1.0,1.0);
    glVertex3f( - Win_Size/3.0,Win_Size/3.0,Win_Size/3.0);
    glTexCoord2f(0.0,1.0);
    glVertex3f( - Win_Size/3.0,Win_Size/3.0, - Win_Size/3.0);
glEnd();
glBindTexture(GL_TEXTURE_2D,ImageIDs[4]);       //绑定到上表面和下表面
glBegin(GL_POLYGON);
    glNormal3f(0.0,1.0,0.0);
    glTexCoord2f(0.0f,0.0f);
    glVertex3f( - Win_Size/3.0,Win_Size/3.0, - Win_Size/3.0);
    glTexCoord2f(0.5,0.0);
    glVertex3f( - Win_Size/3.0,Win_Size/3.0,Win_Size/3.0);
    glTexCoord2f(0.5,1.0);
    glVertex3f(Win_Size/3.0,Win_Size/3.0,Win_Size/3.0);
    glTexCoord2f(0.0,1.0);
    glVertex3f(Win_Size/3.0,Win_Size/3.0, - Win_Size/3.0);
glEnd();
glBegin(GL_POLYGON);
```

```
            glNormal3f(0.0, - 1.0,0.0);
            glTexCoord2f(0.5f,0.0f);
            glVertex3f( - Win_Size/3.0, - Win_Size/3.0, - Win_Size/3.0);
            glTexCoord2f(1.0,0.0);
            glVertex3f(Win_Size/3.0, - Win_Size/3.0, - Win_Size/3.0);
            glTexCoord2f(1.0,1.0);
            glVertex3f(Win_Size/3.0, - Win_Size/3.0,Win_Size/3.0);
            glTexCoord2f(0.5,1.0);
            glVertex3f( - Win_Size/3.0, - Win_Size/3.0,Win_Size/3.0);
        glEnd();
    }
```

10.4.3 纹理透明

纹理也可以是透明的,使用透明纹理映射的同时,还会使得被贴图的三维实体也变得透明。实现透明纹理映射可以采用多种技术,最直观的透明纹理映射是将一幅像素图像加上一定的透明度,即将 24 位图像扩展成 32 位,然后再采用混合技术实现透明。该方法复杂度较高,效率也较低,但效果最好。还有一种方法是利用纹理映射的技术实现,这时需要调用 GLU 库中的 gluBuild2DMipmaps()命令建立二维 mipmap 图像。

所谓 mipmap,是指具有多个明细等级的纹理图像。与其他实体一样,对于带有纹理的实体,可能会从不同距离观察。在动态场景中,当带有纹理的实体远离视点时,纹理图像随实体变小而变小。为此,OpenGL 必须对纹理图像进行相应的滤波,使其大小适合粘贴到实体上,同时避免产生晃动、闪烁和闪光等人工雕琢的现象。为了避免这种人工雕琢的效果,可以指定一系列预先通过滤波生成的、分辨率递减的纹理图像,即 mipmap。

在使用 mipmap 技术时,OpenGL 会根据实体的大小(以像素为单位)自动确定使用哪个纹理图像。而要使用 mipmap 技术,则必须提供从最小尺寸到 1×1、大小为 2 的整数次幂的各种纹理图像。gluBuild2DMipmaps()命令除了获取输入的图像,也可以利用该图像创建 mipmap 图像,以后通过调用 gluScaleImage()命令生成所有级别的 mipmap 图像。gluBuild2DMipmaps()命令的原型为:

```
void gluBuild2DMipmaps(GLenum target, GLint components, GLint width, GLint height, GLenum
format,GLenum type,const void * data);
```

其中参数 target 必须为 TL_TEXTURE_2D; 参数 components 为纹理颜色组分数,其值为 1、2、3 或 4; 参数 width 和 height 分别为纹理图像的宽度和高度; 参数 format 为图像的格式,其取值如表 10.3-2 所示; 参数 type 为图像的数据类型,其值如表 10.3-1 所示; 参数 data 为图像数据的首地址。

因为 gluBuild2DMipmap()也用于图像的加载获取,所以在使用时,只需用其代替原来的图像加载命令 glTexImage2D()即可。

在实现透明时,仍然需要使用混合功能,并调用混合函数 glBlendFunc()设置混合方法。

在程序 OpenGL001 中举例实现纹理透明时,可以对本章 10.4.2 节创建的多纹理对象设置透明纹理。首先,在应用程序工具栏或者菜单栏中设置一个纹理透明的图标标识,通过

类向导在视图类中创建该标识的消息映射函数，例如为OnImageMapBlend()，在该函数中启用混合功能，并调用混合函数glBlendFunc()设置混合方法。代码参考如下：

```
void COpenGL001View::OnImageMapBlend() {
    glEnable(GL_BLEND);                                    //启用混合
    glShadeModel(GL_SMOOTH);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);    //设置混合
    Invalidate();
}
```

在绘图函数RenderScene()中绘制多纹理对象的代码部分，首先调用glColor()命令将透明度Alpha参数值设置为小于1，例如设为0.4，这样即使图像本身不具有透明度，但是单击纹理透明命令后，多纹理图像之间仍然实现了透明。在绘图函数RenderScene()中的绘制多纹理对象部分加入的代码为：

```
glColor4f(1.0f,1.0f,1.0f,0.4f);
```

执行应用程序，创建一个多纹理对象的立方体纹理贴图，然后，单击纹理透明命令，立方体纹理贴图变为透明的，效果如图10.4-1所示。



图10.4-1 纹理透明效果

上述的纹理透明代码中并没有使用到本节所提到的mipmap技术，因为纹理加载是借用的纹理对象中的代码，若使用mipmap技术纹理透明，只需把其中装载纹理图像的代码glTexImage2D()命令用gluBuild2DMipmap()命令替换即可。

10.4.4 一维纹理

OpenGL的纹理映射主要采用的是二维纹理，即二维图像，但同时也支持一维纹理和三维纹理，其中三维纹理在较高的版本才能实现。一维纹理与二维纹理的准备步骤基本相同，即设置纹理参数、设置纹理环境、载入纹理图像和指定纹理坐标等。

在设置纹理参数时，glTexParameter()命令中对应的纹理对象参数应改为一维纹理对象的枚举常量GL_TEXTURE_1D。纹理环境的设置和二维纹理完全相同。

在载入一维纹理图像时，应调用glTexImage1D()命令，该命令的参数与glTexImage2D()中的参数意义相同，只是缺少height参数，参数target取值为GL_TEXTURE_1D。该命令将一幅一维图像载入内存，以供映射使用。

指定一维纹理图像的坐标除了使用glTexCoord1X()命令显示分配纹理坐标外，还可以调用glTexGen()命令让OpenGL自动生成纹理坐标，该命令的原型为：

```
void glTexGenX(GLenum coord,GLenum pname,Type param);
```

其中，X可以是d、f、i或者dv、fv、iv，分别表示参数param的数据类型为双精度型、实型、整型以及这些类型的数组；参数coord表示需要自动生成一维纹理的坐标方向，取值为

GL_S、GL_T、GL_R 或者 GL_Q；参数 pname 为纹理坐标生成函数名，对于简单参数，pname 取值必须为 GL_TEXTURE_GEN_MODE，对于数值参数，pname 取值为 GL_TEXTURE_GEN_MODE、GL_OBJECT_PLANE 或者 GL_EYE_PLANE；参数 param 为生成函数的参数，取值为 GL_OBJECT_LINEAR、GL_EYE_LINEAR、GL_SPHERE_MAP、GL_REFLECTION_MAP 或者 GL_NORMAL_MAP，这些值决定了使用哪个函数生成纹理坐标。如果 param 是一个指向数组的指针，那么这个数组指定了纹理生成函数的参数。

例如，当指定 GL_TEXTURE_GEN_MODE 和 GL_OBJECT_LINEAR 时，纹理生成函数就是物体顶点坐标(x_0，y_0，z_0，w_0)的线性组合：

$$g = p_1x_0 + p_2y_0 + p_3z_0 + p_4w_0$$

其中，g 为生成的坐标；p_i 为参数 param 提供的 4 个值，此时 pname 参数设置为 GL_OBJECT_PLANE；x_0、y_0、z_0、w_0 为向量的齐次坐标。如果 $p_1,\cdots,p_4$ 已经进行了正确的规范化，这个函数将会给出从这个顶点到一个平面的距离。例如，$p_2=p_3=p_4=0$ 并且 $p_1=1$，那么这个函数给出顶点和 $x=0$ 平面之间的距离。当物体在 xy 平面上时，纹理的轮廓线垂直于物体坐标系的 x 轴。

设置自动生成纹理坐标后，还需利用 GL_TEXTURE_GEN_S 为参数调用 glEnable()命令，启用纹理坐标。

和二维纹理映射一样，也需利用 GL_TEXTURE_1D 为参数调用 glEnable()命令，启用一维纹理映射。

在程序 OpenGL001 中举例实现一维纹理时，可以首先在视图类文件中定义一个一维纹理的宏：

```
#define TEXTURE_MAP_1D    33
```

在视图类中，定义一个一维纹理对象变量：

```
GLuint Texture_1D;                                        //一维纹理
```

然后，在应用程序工具栏或者菜单栏中设置一个一维纹理的图标标识，通过类向导在视图类中创建该标识的消息映射函数，例如为 OnTexture1d()，在该函数中创建一个简单的一维位图图像，产生和绑定一维纹理对象 ID，以及加载位图图像、设置一维纹理参数和纹理环境，并设置自动生成纹理坐标。代码参考如下：

```
void COpenGL001View::OnTexture1d() {                      //一维纹理
    //创建一维纹理位图图像
    GLbyte Image_tex_1D[4 * 32];
    int j;
    for(j = 0;j < 32;j++){
        Image_tex_1D[4 * j] = GLubyte((j <= 4)?255:0);
        Image_tex_1D[4 * j + 1] = GLubyte((j > 4)?255:0);
        Image_tex_1D[4 * j + 2] = GLubyte(0);
        Image_tex_1D[4 * j + 3] = GLubyte(255);
    }
    //创建和绑定一维纹理对象
    glGenTextures(1,&Texture_1D);
```

```
    glBindTexture(GL_TEXTURE_1D,Texture_1D);
    //设置纹理参数
    glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    //加载纹理图像
    glTexImage1D(GL_TEXTURE_1D,0,GL_RGBA,32,0,GL_RGBA,GL_UNSIGNED_BYTE,Image_tex_1D);
    //设置纹理环境
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
    //自动生成纹理坐标
    GLfloat fzero[] = {1.0,0.0,0.0,0.0};
    glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
    glTexGenfv(GL_S,GL_OBJECT_PLANE,fzero);
    glEnable(GL_TEXTURE_GEN_S);                    //启用纹理坐标
    glEnable(GL_TEXTURE_1D);                       //启用一维纹理映射
    glEnable(GL_AUTO_NORMAL);                      //设置法向量
    glEnable(GL_NORMALIZE);                        //设置法向量规范化
    glFrontFace(GL_CW);                            //设置顶点走向
    m_flag = TEXTURE_MAP_1D;                       //设置一维纹理映射宏
    InitOperation();
}
```

在绘图函数 RenderScene()中加入映射一维纹理图像到某个形体表面的代码，由于上述一维纹理已经设置完毕，该纹理可映射到任何表面。本例将其映射到一个简单的平面上，代码放在最后一行代码 glPopMatrix()之前，参考如下：

```
if(m_flag == TEXTURE_MAP_1D){
    glBindTexture(GL_TEXTURE_1D,Texture_1D);
    glBegin(GL_POLYGON);
        glNormal3f(0.0,0.0,1.0);
        glVertex3f( - Win_Size/3.0, - Win_Size/3.0,0.0);
        glVertex3f(Win_Size/3.0, - Win_Size/3.0,0.0);
        glVertex3f(Win_Size/3.0,Win_Size/3.0,0.0);
        glVertex3f( - Win_Size/3.0,Win_Size/3.0,0.0);
    glEnd();
}
```

运行程序，单击一维纹理命令后，可以在一个平面上绘制一幅一维纹理图像。

10.4.5 球体纹理

在纹理坐标生成函数 glTexGen()中，当其参数取 GL_TEXTURE_GEN_MODE 和 GL_SPHERE_MAP 时，OpenGL 计算纹理坐标的方式相当于使物体看上去相对当前纹理贴图的反射，这种纹理生成模式又称为球体纹理映射(贴图)。球体纹理映射设置方法如下：

```
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
```

并通过调用 glEnable()命令对 S、T 纹理坐标启用纹理生成：

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

当纹理坐标生成功能启用后，对 glTexCoord()的任何调用都将被忽略，OpenGL 自动计算每个顶点的纹理坐标。如果利用 glTexCoord()指定纹理坐标，那么，必须禁用纹理坐标生成功能：

```
glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
```

同理，对 R 和 Q 纹理坐标生成的启用和禁用设置方法也同上。

在程序 OpenGL001 中举例实现球体纹理时，可以首先在视图类文件中定义一个球体纹理的宏：

```
#define TEXTURE_SPHERE    36
```

在视图类中，定义一个球体纹理对象数组，一个为被反射的纹理对象，一个为反射的球体纹理对象，它们加载同一幅图像，以便产生反射的效果：

```
GLuint ImageSphere[2];;                              //球体纹理对象
```

然后，在应用程序工具栏或者菜单栏中设置一个球体纹理的图标标识，通过类向导在视图类中创建该标识的消息映射函数，例如为 OnTextrureSphere()，在该函数中创建纹理对象以及加载位图图像，设置纹理参数和纹理环境，设置球体纹理坐标，启用纹理映射等。代码参考如下：

```
void COpenGL001View::OnTextrureSphere() {
CFileDialog hFileDlg(true,NULL,NULL,OFN_FILEMUSTEXIST| OFN_READONLY| OFN_PATHMUSTEXIST, TEXT
("bmp 文件(*.bmp)|*.bmp|"), NULL);                     //打开图像
    if(hFileDlg.DoModal() == IDOK){
        AUX_RGBImageRec * m_image;
        glPixelStorei(GL_UNPACK_ALIGNMENT,1);
        m_image = auxDIBImageLoad(hFileDlg.GetPathName());
        m_iWidth = m_image->sizeX;
        m_iHeight = m_image->sizeY;
        m_pImage = m_image->data;
        glGenTextures(2,ImageSphere);                  //创建纹理对象
        for(int i = 0;i < 2;i++){                      //纹理对象加载和设置同一图像
            glBindTexture(GL_TEXTURE_2D,ImageSphere[i]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glTexImage2D(GL_TEXTURE_2D,0,3,m_iWidth,m_iHeight,0,GL_RGB,GL_UNSIGNED_BYTE,m_pImage);
            m_iMode = GL_REPLACE;
        }
        m_flag = TEXTURE_SPHERE;
        InitOperation();
        glEnable(GL_TEXTURE_2D);                       //启用纹理
```

```
        //启用球体纹理
        glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
        glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
        glEnable(GL_DEPTH_TEST);                        //启用深度检测
    }
}
```

在绘图函数 RenderScene()中，为了实现球体纹理的效果，需要实现两部分纹理映射，一个是被反射的纹理映射，一个是反射的球体纹理。可以将整个绘图窗口背景设为被反射的纹理，由于背景是固定的，因此，绘制这个背景纹理的代码放在 RenderScene()函数的最前面，不进行动画和移动。代码如下：

```
if(m_flag == TEXTURE_SPHERE){
    glPushMatrix();
    int cx,cy;
    cx = (int)winWidth * 2;
    cy = (int)winHeight * 2;
    ::glViewport(0, 0, cx, cy);
    if(aspect_ratio < 1){                               //设置绘图窗口大小
        winWidth = Win_Size;
        winHeight = Win_Size/aspect_ratio;
    }
    else{
        winWidth = Win_Size * aspect_ratio;
        winHeight = Win_Size;
    }
    glBindTexture(GL_TEXTURE_2D,ImageSphere[0]);        //绑定纹理对象
    glDisable(GL_TEXTURE_GEN_S);                        //禁用纹理坐标生成
    glDisable(GL_TEXTURE_GEN_T);                        //禁用纹理坐标生成
    glDepthMask(GL_FALSE);                              //背景绘图无深度缓冲区
    glBegin(GL_QUADS);
        glTexCoord2f(0.0,0.0);
        glVertex2f( - winWidth, - winHeight);
        glTexCoord2f(1.0,0.0);
        glVertex2f(winWidth, - winHeight);
        glTexCoord2f(1.0,1.0);
        glVertex2f(winWidth,winHeight);
        glTexCoord2f(0.0,1.0);
        glVertex2f( - winWidth,winHeight);
    glEnd();
    winWidth = cx/2.0;
    winHeight = cy/2.0;
    glPopMatrix();
}
```

对于反射的球体纹理，可以映射在一个圆环上，圆环可以动画展示和移动，因此，这部分代码放在 RenderScene()函数最后的代码“glPopMatrix();”之前，参考如下：

```
if(m_flag == TEXTURE_SPHERE){
    glEnable(GL_TEXTURE_GEN_S);                         //启用纹理坐标生成
```

```
    glEnable(GL_TEXTURE_GEN_T);                         //启用纹理坐标生成
    glDepthMask(GL_TRUE);
    glBindTexture(GL_TEXTURE_2D,ImageSphere[1]);        //绑定纹理对象
    glutSolidTorus(Win_Size/4.0,Win_Size/3.0,30,30);
}
```

执行程序,单击球体纹理,加载一个图像后,移动或者动画展示创建的圆环,这时圆环的表面纹理具有反射背景图像的效果。

10.4.6 立方图纹理及天空盒绘制和表面反射

在纹理坐标生成函数 glTexGen()中,当纹理坐标生成模式为 GL_REFLECTION_MAP 和 GL_NORMAL_MAP 时,OpenGL 使用了一种新的纹理环境:立方图纹理(cube map)。立方图纹理被认为是单个纹理,但它实际上是由 6 个纹理所组成的一个立方体的 6 个面。这 6 个面代表从不同方向(前、后、左、右、上和下)观察所看到的图像。使用 GL_REFLECTION_MAP 纹理坐标生成模式,也可以创建精确的反射表面。

立方图纹理的 6 个面要分别加载图像,这 6 个面对应的目标参数分别为:

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X       右面
GL_TEXTURE_CUBE_MAP_NEGATIVE_X       左面
GL_TEXTURE_CUBE_MAP_POSITIVE_Y       顶面
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y       下面
GL_TEXTURE_CUBE_MAP_POSITIVE_Z       后面
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z       前面
```

例如,加载右面的纹理贴图函数代码为:

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X,0,iComponents,iWidth, iHeight, 0, eFormat, GL_
UNSIGNED_BYTE, pBytes);
```

为了启用立方图纹理,应以 GL_TEXTURE_CUBE_MAP 为参数调用 glEnable()函数。在使用立方图纹理对象时,在 glBindTexture()函数中也使用 GL_TEXTURE_CUBE_MAP 参数值。如果 GL_TEXTURE_CUBE_MAP 和 GL_TEXTURE_2D 两个模式都被启用,则 GL_TEXTURE_CUBE_MAP 被优先启用。

对立方图进行纹理参数设置、环境参数设置以及纹理坐标设置,影响的是立方图纹理的所有 6 幅图像,因此,可以统一设置,例如:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MAG_FILTER,GL_ LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_MIN_FILTER,GL_ LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_S,L_CLAMP_TO _EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_T,GL_CLAMP_ TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_R,GL_CLAMP _TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_GENERATE_MIPMAP, GL_TRUE);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
```

立方图纹理的纹理坐标和真正的3D纹理不同,S、T和R纹理坐标表示一个从中心指向纹理贴图的向量,这个向量与立方图纹理的其中一面相交,然后对相交四周的纹理单元进行采样,根据纹理创建经过过滤的颜色值。

立方图纹理常见的用途是绘制天空盒和创建对周围景物的反射。天空盒没什么特别的,整个场景就像一个大立方体盒子,它的上面是天空的图像,利用立方体纹理绘制这个大立方体的六个面,每个面绘制成GL_QUADS组成的图形,对于每个顶点,使用glTexCoord3f()手动设置纹理坐标,并指定顶点向量。

因为要手动设置天空盒中顶点的纹理坐标,因此要禁用纹理坐标的自动生成功能:

```
glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
glDisable(GL_TEXTURE_GEN_R);
```

当创建物体对立方图纹理图像的反射功能时,需要将禁用的纹理坐标自动生成功能启用,并设置坐标生成模式为GL_REFLECTION_MAP:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
```

在程序中实现立方图纹理及天空盒绘制和物体表面反射功能时,首先需要6幅图像,本节参考《OpenGL超级宝典(第4版)》相关章节,从相应网站下载了对应的6幅天空盒图像文件,名字分别为pos_x.tga,neg_x.tga,pos_y.tga,neg_y.tga,pos_z.tga,neg_z.tga,因图像为tga格式,因此,程序中需要读取该格式文件。由于天空盒和反射功能需要在天空盒内部观察,投影设置为透视模式较合适,故本节单独建立了一个控制台应用程序,将6幅天空盒图像文件放在该程序目录下,在源代码中加载立方图纹理时,加载该6幅图像,并绘制到天空盒表面,立方图反射功能通过绘制的圆环来呈现。具体源代码如下(该代码中也包含了tga格式图片文件的读取方法):

```
#define GLUT_DISABLE_ATEXIT_HACK
#include <stdio.h>
#include <gl/glut.h>
#include <gl/glaux.h>
#include <gl/glext.h>
#pragma comment(lib,"glaux")
const char *szCubeFaces[6] = { "pos_x.tga", "neg_x.tga", "pos_y.tga", "neg_y.tga", "pos_z.
tga", "neg_z.tga" };
GLenum cube[6] = {  GL_TEXTURE_CUBE_MAP_POSITIVE_X,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
                    GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
                    GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_Z };
#pragma pack(1)
typedef struct{
    GLbyte identsize;               //Size of ID field that follows header (0)
    GLbyte colorMapType;            //0 = None, 1 = paletted
```

```
    GLbyte imageType;                    //0 = none, 1 = indexed, 2 = rgb, 3 = grey, +8 = rle
    unsigned short  colorMapStart;       //First colour map entry
    unsigned short  colorMapLength;      //Number of colors
    unsigned char   colorMapBits;        //bits per palette entry
    unsigned short  xstart;              //image x origin
    unsigned short  ystart;              //image y origin
    unsigned short  width;               //width in pixels
    unsigned short  height;              //height in pixels
    GLbyte bits;                         //bits per pixel (8 16, 24, 32)
    GLbyte descriptor;                   //image descriptor
} TGAHEADER;
#pragma pack(8)
GLint gltWriteTGA(const char *szFileName){
    FILE *pFile;                         //File pointer
    TGAHEADER tgaHeader;                 //TGA file header
    unsigned long lImageSize;            //Size in bytes of image
    GLbyte *pBits = NULL;                //Pointer to bits
    GLint iViewport[4];                  //Viewport in pixels
    GLenum lastBuffer;                   //Storage for the current read buffer setting
    //Get the viewport dimensions
    glGetIntegerv(GL_VIEWPORT, iViewport);
    //How big is the image going to be (targas are tightly packed)
    lImageSize = iViewport[2] * 3 * iViewport[3];
    //Allocate block. If this doesn't work, go home
    pBits = (GLbyte *)malloc(lImageSize);
    if(pBits == NULL)
        return 0;
    //Read bits from color buffer
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
    glPixelStorei(GL_PACK_ROW_LENGTH, 0);
    glPixelStorei(GL_PACK_SKIP_ROWS, 0);
    glPixelStorei(GL_PACK_SKIP_PIXELS, 0);
    //Get the current read buffer setting and save it. Switch to
    //the front buffer and do the read operation. Finally, restore
    //the read buffer state
    glGetIntegerv(GL_READ_BUFFER, (GLint *)&lastBuffer);
    glReadBuffer(GL_FRONT);
    glReadPixels(0, 0, iViewport[2], iViewport[3], GL_BGR_EXT, GL_UNSIGNED_BYTE, pBits);
    glReadBuffer(lastBuffer);
    //Initialize the Targa header
    tgaHeader.identsize = 0;
    tgaHeader.colorMapType = 0;
    tgaHeader.imageType = 2;
    tgaHeader.colorMapStart = 0;
    tgaHeader.colorMapLength = 0;
    tgaHeader.colorMapBits = 0;
    tgaHeader.xstart = 0;
    tgaHeader.ystart = 0;
    tgaHeader.width = iViewport[2];
    tgaHeader.height = iViewport[3];
    tgaHeader.bits = 24;
```

```
    tgaHeader.descriptor = 0;
    //Do byte swap for big vs little endian
#ifdef __APPLE__
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapStart);
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapLength);
    LITTLE_ENDIAN_WORD(&tgaHeader.xstart);
    LITTLE_ENDIAN_WORD(&tgaHeader.ystart);
    LITTLE_ENDIAN_WORD(&tgaHeader.width);
    LITTLE_ENDIAN_WORD(&tgaHeader.height);
#endif
    //Attempt to open the file
    pFile = fopen(szFileName, "wb");
    if(pFile == NULL){
        free(pBits);                //Free buffer and return error
        return 0;
        }
    //Write the header
    fwrite(&tgaHeader, sizeof(TGAHEADER), 1, pFile);
    //Write the image data
    fwrite(pBits, lImageSize, 1, pFile);
    //Free temporary buffer and close the file
    free(pBits);
    fclose(pFile);
    //Success!
    return 1;
}
GLbyte * gltLoadTGA (const char * szFileName, GLint * iWidth, GLint * iHeight, GLint *
iComponents, GLenum * eFormat){
    FILE * pFile;                   //File pointer
    TGAHEADER tgaHeader;            //TGA file header
    unsigned long lImageSize;       //Size in bytes of image
    short sDepth;                   //Pixel depth;
    GLbyte * pBits = NULL;          //Pointer to bits
    //Default/Failed values
    * iWidth = 0;
    * iHeight = 0;
    * eFormat = GL_BGR_EXT;
    * iComponents = GL_RGB8;
    //Attempt to open the fil
    pFile = fopen(szFileName, "rb");
    if(pFile == NULL)
        return NULL;
    //Read in header (binary)
    fread(&tgaHeader, 18/* sizeof(TGAHEADER) */, 1, pFile);
    //Do byte swap for big vs little endian
#ifdef __APPLE__
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapStart);
    LITTLE_ENDIAN_WORD(&tgaHeader.colorMapLength);
    LITTLE_ENDIAN_WORD(&tgaHeader.xstart);
    LITTLE_ENDIAN_WORD(&tgaHeader.ystart);
    LITTLE_ENDIAN_WORD(&tgaHeader.width);
```

```
    LITTLE_ENDIAN_WORD(&tgaHeader.height);
#endif
    //Get width, height, and depth of texture
    *iWidth = tgaHeader.width;
    *iHeight = tgaHeader.height;
    sDepth = tgaHeader.bits / 8;
    //Put some validity checks here. Very simply, I only understand
    //or care about 8, 24, or 32 bit targa's.
    if(tgaHeader.bits != 8 && tgaHeader.bits != 24 && tgaHeader.bits != 32)
        return NULL;
    //Calculate size of image buffer
    lImageSize = tgaHeader.width * tgaHeader.height * sDepth;
    //Allocate memory and check for success
    pBits = (GLbyte*)malloc(lImageSize * sizeof(GLbyte));
    if(pBits == NULL)
        return NULL;
    //Read in the bits
    //Check for read error. This should catch RLE or other
    //weird formats that I don't want to recognize
    if(fread(pBits, lImageSize, 1, pFile) != 1){
        free(pBits);
        return NULL;
        }
    //Set OpenGL format expected
    switch(sDepth){
        case 3:                         //Most likely case
            *eFormat = GL_BGR_EXT;
            *iComponents = GL_RGB8;
            break;
        case 4:
            *eFormat = GL_BGRA_EXT;
            *iComponents = GL_RGBA8;
            break;
        case 1:
            *eFormat = GL_LUMINANCE;
            *iComponents = GL_LUMINANCE8;
            break;
        };
    //Done with File
    fclose(pFile);
    //Return pointer to image data
    return pBits;
}
GLuint textureObjects;                  //纹理对象
GLfloat x_t;                            //移动量
GLfloat m_rAngle;                       //旋转角度
void SetupRC(){
    GLbyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    int i;
```

```
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
    //绑定立方图映射对象
    glGenTextures(1, &textureObjects);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureObjects);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
for(i = 0; i < 6; i++){      //加载纹理映射图像(6 幅)
pBytes = gltLoadTGA(szCubeFaces[i],&iWidth,&iHeight,&iComponents, &eFormat);
glTexImage2D(cube[i],0,iComponents,iWidth,iHeight,0,eFormat, GL_UNSIGNED_BYTE, pBytes);
        free(pBytes);
       }
    x_t = 0.0;
    m_rAngle = 0.0;
}
//绘制天空盒
void DrawSkyBox(void){
    GLfloat fExtent = 15.0f;
    glBegin(GL_QUADS);
        //Negative X 左面
        glTexCoord3f(-1.0f, -1.0f, 1.0f);
        glVertex3f(-fExtent, -fExtent, fExtent);
        glTexCoord3f(-1.0f, -1.0f, -1.0f);
        glVertex3f(-fExtent, -fExtent, -fExtent);
        glTexCoord3f(-1.0f, 1.0f, -1.0f);
        glVertex3f(-fExtent, fExtent, -fExtent);
        glTexCoord3f(-1.0f, 1.0f, 1.0f);
        glVertex3f(-fExtent, fExtent, fExtent);
        //Postive X 右面
        glTexCoord3f(1.0f, -1.0f, -1.0f);
        glVertex3f(fExtent, -fExtent, -fExtent);
        glTexCoord3f(1.0f, -1.0f, 1.0f);
        glVertex3f(fExtent, -fExtent, fExtent);
        glTexCoord3f(1.0f, 1.0f, 1.0f);
        glVertex3f(fExtent, fExtent, fExtent);
        glTexCoord3f(1.0f, 1.0f, -1.0f);
        glVertex3f(fExtent, fExtent, -fExtent);
        //Negative Z 前面
        glTexCoord3f( -1.0f, -1.0f, -1.0f);
        glVertex3f(-fExtent, -fExtent, -fExtent);
        glTexCoord3f(1.0f, -1.0f, -1.0f);
        glVertex3f(fExtent, -fExtent, -fExtent);
```

```
        glTexCoord3f(1.0f, 1.0f, -1.0f);
        glVertex3f(fExtent, fExtent, -fExtent);
        glTexCoord3f(-1.0f, 1.0f, -1.0f);
        glVertex3f(-fExtent, fExtent, -fExtent);
        //Positive Z 后面
        glTexCoord3f(1.0f, -1.0f, 1.0f);
        glVertex3f(fExtent, -fExtent, fExtent);
        glTexCoord3f(-1.0f, -1.0f, 1.0f);
        glVertex3f(-fExtent, -fExtent, fExtent);
        glTexCoord3f(-1.0f, -1.0f, 1.0f);
        glVertex3f(-fExtent, fExtent, fExtent);
        glTexCoord3f( 1.0f, 1.0f, 1.0f);
        glVertex3f(fExtent, fExtent, fExtent);
        //Positive Y 上面
        glTexCoord3f(-1.0f, 1.0f, 1.0f);
        glVertex3f(-fExtent, fExtent, fExtent);
        glTexCoord3f( -1.0f, 1.0f, -1.0f);
        glVertex3f(-fExtent, fExtent, -fExtent);
        glTexCoord3f(1.0f, 1.0f, -1.0f);
        glVertex3f(fExtent, fExtent, -fExtent);
        glTexCoord3f( 1.0f, 1.0f, 1.0f);
        glVertex3f(fExtent, fExtent, fExtent);
        //Negative Y 下面
        glTexCoord3f(-1.0f, -1.0f, -1.0f);
        glVertex3f(-fExtent, -fExtent, -fExtent);
        glTexCoord3f(-1.0f, -1.0f, 1.0f);
        glVertex3f(-fExtent, -fExtent, fExtent);
        glTexCoord3f(1.0f, -1.0f, 1.0f);
        glVertex3f(fExtent, -fExtent, fExtent);
        glTexCoord3f( 1.0f, -1.0f, -1.0f);
        glVertex3f(fExtent, -fExtent, -fExtent);
    glEnd();
}
void RenderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
        //绘制天空盒
        glDisable(GL_TEXTURE_GEN_S);
        glDisable(GL_TEXTURE_GEN_T);
        glDisable(GL_TEXTURE_GEN_R);
        DrawSkyBox();
        //反射
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
        glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);
        glEnable(GL_TEXTURE_GEN_R);
```

```
        glPushMatrix();
            glTranslatef(x_t, 0, -3.0f);
            glRotatef(m_rAngle,1.0f,0.0f,0.0f);
            glutSolidTorus(0.40,0.45,30,30);    //绘制圆环
        glPopMatrix();
    glPopMatrix();
    glutSwapBuffers();
}
void SpecialKeys(int key, int x, int y) {           //键盘操作
   if(key == GLUT_KEY_UP)
      m_rAngle += 0.1;
    if(key == GLUT_KEY_DOWN)
        m_rAngle -= 0.1;
    if(key == GLUT_KEY_LEFT)
        x_t -= 0.1;
    if(key == GLUT_KEY_RIGHT)
        x_t += 0.1;
    glutPostRedisplay();
}
void ChangeSize(int w, int h){                      //窗口改变
    GLfloat fAspect;
    if(h == 0)
        h = 1;
    glViewport(0, 0, w, h);
    fAspect = (GLfloat)w / (GLfloat)h;
    //Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    //Set the clipping volume
    gluPerspective(35.0f, fAspect, 1.0f, 2000.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc, char * argv[]){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800,600);
    glutCreateWindow("OpenGL 立方图纹理映射");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    glutSpecialFunc(SpecialKeys);
    SetupRC();
    glutMainLoop();
    return 0;
}
```

运行程序,效果如图 10.4-2 所示。可以通过键盘的上下键旋转圆环和左右键移动圆环来观察立方图反射的效果。



图 10.4-2　立方图纹理天空盒及反射

10.5　OpenGL 曲线曲面技术

10.5.1　绘制二次曲面

OpenGL 不仅支持直线和平面几何图形对象，也支持曲线和曲面对象，但从本质上讲，OpenGL 中的曲线曲面仍然是由直线和平面构成的，组成曲线的直线段越短，则曲线越平顺；组成曲面的平面片越小，则曲面越光滑。在曲线曲面中，像球体、圆柱体和圆锥体等一类几何对象可以通过二次方程表示，所以，又称之为二次曲面实体。OpenGL 的 GLU 工具函数库定义了一些二次曲面实体的绘制命令，可以直接绘制相关的二次曲面实体。

利用 GLU 库绘制二次曲面的一般步骤如下：

(1) 定义一个 GLUquadricObj 类型的指针变量，并通过调用 gluNewQuadric()命令来创建和初始化一个二次曲面对象为该指针变量赋值，代码如下：

```
GLUquadricObj * pObj;
pObj = gluNewQuadric();
```

(2) 设置二次曲面对象的绘图参数。使用下面的四个函数来设置。

设置曲面绘制方式的函数为：

```
void gluQuadricDrawStyle(GLUquadricObj * pObj,GLenum drawStyle);
```

其中，参数 pObj 即为步骤(1)创建的二次曲面对象指针；drawStyle 用于设置图形的显示风格，取如下值：

GLU_FILL——以实体渲染形式绘制显示

GLU_LINE——以线框形式绘制显示

GLU_POINT——以一组顶点集合的形式显示

GLU_SILHOUETTE——以轮廓形式显示

指定曲面上法线指向表面外部还是指向表面内部的函数为：

```
void gluQuadricOrientation(GLUquadricObj *pObj,GLenum orientation);
```

其中，参数 pObj 的意义和取值同上；orientation 可以是 GLU_OUTSIDE 或 GLU_INSIDE。默认情况下，二次方程表面是根据逆时针方向进行环绕，它的正面是表面的外部。对于球体和圆柱体，外表面朝向观察者；对于圆盘，正面是沿 z 轴正方向的那个面。

指定二次方程表面几何图形在生成时是否带有表面法线的函数为：

```
void gluQuadricNormals(GLUquadricObj *pObj,GLenum normals);
```

其中，参数 normals 的值为 GL_NONE(没有法线)、GL_SMOOTH(为图像每个顶点都生成法向量，为默认值)或 GL_FLAT(为组成曲面的每个小平面生成一个法向量)。

当为二次曲面申请设置纹理坐标时，需调用下面的函数实现：

```
void gluQuadricTexture(GLUquadricObj *pObj,GLenum textureCoords);
```

其中，参数 textureCoords 取值为 GL_TRUE 或 GL_FALSE。当设置纹理坐标时，纹理沿球体或圆柱体均匀环绕，对于圆盘，纹理中心应用到圆盘中心，纹理边缘应用到圆盘的边缘。

(3) 调用二次曲面绘制命令绘制相应的二次曲面对象。GLU 库实际上仅提供了四种二次曲面对象：球体、圆盘、扇形盘和圆柱。

绘制球体的函数为：

```
void gluSphere(GLUquadricObj *pObj,GLdouble radius,GLint slices,GLint stacks);
```

其中，参数 pObj 的意义和取值同上文；radius 为球的半径；slices 和 stacks 分别表示绘制球时划分的经线方向的分段数和纬线方向的分段数，数值越大，球面越光滑。该绘制球体的函数与 GLUT 库中绘制球体的命令基本相同，只是多了一个二次曲面的对象指针变量。

绘制圆柱的函数为：

```
void gluCylinder(GLUquadricObj *pObj,GLdouble baseRadius,GLdouble topRadius, GLdouble
height,GLint slices,GLint stacks);
```

其中，baseRadius 为圆柱的底部半径，topRadius 为顶部半径，height 为圆柱的高，其他参数和绘制球的函数的参数类似。

绘制圆盘的函数为：

```
void gluDisk(GLUquadricObj *pObj,GLdouble innerRadius,GLdouble outerRadius,GLint slices,
GLint loops);
```

其中，参数 innerRadius 为圆盘内半径，outerRadius 为圆盘外半径，loops 为径向分段数，其他参数同上文。如果内半径为 0，则表示是一个实心圆盘。

绘制不完整圆盘即扇形盘的函数为：

```
void gluPartialDisk(GLUquadricObj *pObj,GLdouble innerRadius,GLdouble outerRadius, GLint
slices,GLint loops,GLdouble startAngle,GLdouble sweepAngle);
```

其中的参数与 gluDisk()函数中同名的参数意义相同；参数 startAngle 为圆盘缺口的开始角，以角度计；sweepAngle 为扫描的角度。该命令绘制一个中心在原点，且垂直于 z 轴的不完整圆盘。

(4) 所有二次曲线对象绘制完毕，调用 glDeleteQuadric()命令将指针所指的二次曲面对象删除。命令原型为：

```
void gluDeleteQuadric(GLUquadricObj *pObj);
```

对 OpenGL 提供的这几种二次曲面实体，可以调用 glScale()命令进行缩放变换，当沿各个坐标方向的比例不相等时，原来的二次曲面实体会发生变形，从而获得需要的造型。由于缩放变换会导致各二次曲面和三维实体预先定义的法向量变成非单位向量，为避免最终的渲染产生明显失真，在具体绘图前，先调用 glEnable()命令启用 GL_NORMALIZE 功能，要求系统将变化后的法向量单位化。

在程序 OpenGL001 中举例实现二次曲面对象时，首先在视图类文件中定义一个二次曲面对象的宏：

```
#define QUADRICOBJ    40
```

然后，在应用程序工具栏或者菜单栏中设置一个二次曲面的图标标识，通过类向导在视图类中创建该图标标识的消息映射函数，在该函数中设置绘图命令标识变量 m_flag= QUADRICOBJ，并通过 InitOperation()初始化设置绘图窗口。

在绘图函数 RenderScene()中加入绘制二次曲面的代码，其中的二次曲面有球、圆柱面、圆锥面、圆盘以及相应变形的形状。代码放在最后一行的 glPopMatrix()之前，参考如下：

```
if(m_flag==QUADRICOBJ){
    GLUquadricObj *pObj;
    pObj=gluNewQuadric();                            //创建和初始化二次曲面对象
    gluQuadricDrawStyle(pObj,GLU_FILL);
    gluQuadricOrientation(pObj,GLU_OUTSIDE);
    gluQuadricNormals(pObj,GL_SMOOTH);
    glEnable(GL_NORMALIZE);                          //启用法向量单位化
    glPushMatrix();
        glTranslatef(-Win_Size/2.0,Win_Size/2.0,0.0);
        glColor3f(1.0f,0.0f,0.0f);
        gluSphere(pObj,Win_Size/4.0,20,20);          //绘制球
    glPopMatrix();
    glPushMatrix();
        glTranslatef(0.0,Win_Size/2.0,0.0);
        glColor3f(1.0f,1.0f,0.0f);
        gluDisk(pObj,Win_Size/6.0,Win_Size/4.0,20,20); //绘制圆盘
    glPopMatrix();
    glPushMatrix();
        glTranslatef(Win_Size/2.0,Win_Size/2.0,0.0);
```

```
        glColor3f(0.0f,1.0f,1.0f);
        gluPartialDisk(pObj,Win_Size/6.0,Win_Size/4.0,20,20,30,300);      //绘制扇形盘
    glPopMatrix();
    glPushMatrix();
        glTranslatef( - Win_Size/2.0,0,0.0);
        glColor3f(1.0f,0.0f,0.0f);
        gluCylinder(pObj,Win_Size/5.0,Win_Size/5.0, Win_Size/5.0,20,20);  //绘制圆柱
    glPopMatrix();
    glPushMatrix();
            glTranslatef(0.0,0.0,0.0);
            glColor3f(1.0f,1.0f,0.0f);
            gluCylinder(pObj,Win_Size/5.0,0.0, Win_Size/5.0,20,20);       //绘制圆锥
    glPopMatrix();
    glPushMatrix();
        glTranslatef(Win_Size/2.0,0.0,0.0);
        glColor3f(0.0f,1.0f,1.0f);
        gluCylinder(pObj,Win_Size/5.0,Win_Size/6.0, Win_Size/5.0,20,20);  //绘制圆锥
    glPopMatrix();
    glPushMatrix();
        glTranslatef( - Win_Size/2.0, - Win_Size/2.0,0.0);
        glColor3f(1.0f,0.0f,0.0f);
        glScalef(0.5f,1.0f,1.0f);
        gluSphere(pObj,Win_Size/4.0,20,20);                               //绘制橄榄球
    glPopMatrix();
    glPushMatrix();
        glTranslatef(0.0, - Win_Size/2.0,0.0);
        glScalef(0.5f,1.0f,1.0f);
        glColor3f(1.0f,1.0f,0.0f);
        gluCylinder(pObj,Win_Size/5.0,Win_Size/5.0, Win_Size/5.0,20,20);  //绘制变形圆柱
    glPopMatrix();
    glPushMatrix();
            glTranslatef(Win_Size/2.0, - Win_Size/2.0,0.0);
            glScalef(1.0f,0.5f,1.0f);
            glColor3f(0.0f,1.0f,1.0f);
            gluDisk(pObj,Win_Size/6.0,Win_Size/4.0,20,20);                //绘制变形圆盘
    glPopMatrix();
    gluDeleteQuadric(pObj);                                             //删除二次曲面对象
}
```

10.5.2 绘制 Bézier 曲线曲面

1. 绘制 Bézier 曲线

OpenGL 除了绘制二次曲面外,还可以绘制 Bézier 样条曲线曲面。在绘制 Bézier 曲线曲面时,首先调用一个求值器函数,将所给的 Bézier 曲线曲面的控制顶点进行映射,并为下一步创建生成曲线曲面上的点做好准备。绘制 Bézier 曲线调用的求值器函数为 glMap1(),该函数原型为:

```
void glMap1d/f(GLenum target,Type u1,Type u2,GLint stride,GLint order,const Type * points);
```

其中，参数 target 为该求值器所产生的数值类型，它可以是表 10.5-1 中的 9 个枚举常量之一，假设控制顶点坐标为(x,y,z)形式，则设置 target 为 GL_MAP1_VERTEX3；参数 u1、u2 为 Bézier 曲线参数变量 u 的下限和上限，要求 u1<u2，理论上，这两个值应分别取值 0 和 1，在 OpenGL 中，可以是任何区间的数值；参数 stride 用于指出控制顶点数组 points 中相邻点之间的数据间距，当控制顶点为(x,y,z)形式时，则间距取 3；参数 order 为曲线控制点的个数；参数 points 为控制顶点数组的首地址。

表 10.5-1 求值器所产生的数值类型

枚举常量	意义
GL_MAP1_VERTEX_3	控制顶点由 3 个实数构成，求值器将产生内部的 glVertex3 值
GL_MAP1_VERTEX_4	控制顶点由 4 个实数构成，求值器将产生内部的 glVertex4 值
GL_MAP1_INDEX	控制顶点是一个表示颜色索引的实数，求值器将产生内部的 glIndex 值
GL_MAP1_COLOR_4	控制顶点由 4 个表示颜色的实数构成，求值器将产生内部的 glColor4 值
GL_MAP1_NORMAL	控制顶点为 3 个表示法向量的实数，求值器将产生 glNormal 值
GL_MAP1_TEXTURE_COORD_1	控制顶点为纹理坐标的单一实数，求值器将产生 glTexCoord1 值
GL_MAP1_TEXTURE_COORD_2	控制顶点为纹理坐标的两个实数，求值器将产生 glTexCoord2 值
GL_MAP1_TEXTURE_COORD_3	控制顶点为纹理坐标的 3 个实数，求值器将产生 glTexCoord1 值
GL_MAP1_TEXTURE_COORD_4	控制顶点为纹理坐标的 4 个实数，求值器将产生 glTexCoord4 值

利用求值器创建映射后，必须利用 GL_MAP1_VERTEX_3 作为参数值调用 glEnable()命令，才能启用求值器生成曲线上的点。

根据求值器创建的曲线上点绘制 Bézier 曲线有两种方法。第一种方法是利用绘制折线的方式绘制曲线：将曲线分成多个区间，将区间点连起来，当分点比较多时，绘出的将是一条光滑的曲线。曲线上区间点通过调用 glEvalCoord1f()命令生成，该命令原型为：

```
void glEvalCoord1X(Type u);
```

其中，X 为 d、f、dv 或者 fv，用于指定参数 u 的数据类型。该函数通过遍历曲线的定义域的参数值生成曲线上的点，因此，参数 u 为曲线定义域内的参数值，在求值器 glMap1()中设定的 u1、u2 区间中。假设 u1、u2 区间为[0,1]，划分 50 个区间，则利用折线段方法绘制 Bézier 的代码可参考如下：

```
glBegin(GL_LINE_STRIP);
    for(int i = 0;i < 50;i++)
        glEvalCoord1f(i/50.0);
glEnd();
```

根据求值器创建的曲线上点绘制 Bézier 曲线的第二种方法是调用 glMapGrid()函数设置一个网格，对于绘制曲线来说，只是在一个参数 u 方向上形成一维网格，函数原型为：

```
void glMapGrid1X(GLint n,Type u1,Type u2);
```

其中，X 为 d 或者 f，用于指定 u1、u2 的数据类型；参数 n 为网格分段数；u1、u2 为参数定义域的下限和上限。

然后，调用 glEvalMesh()函数连接各网格点，产生 Bézier 曲线，函数原型为：

```
void glEvalMesh(GLenum mode,GLint p1,GLint p2);
```

其中，参数 mode 的值可以是 GL_POINT 或者 GL_LINE，取决于想沿这条曲线绘制点还是绘制相连的直线；p1、p2 分别为绘制的起始段和终止段。

利用第二种方法也可以绘制一条 Bézier 曲线。

在程序 OpenGL001 中绘制 Bézier 曲线，可以通过鼠标在屏幕坐标系拾取控制顶点，然后再绘制曲线。首先，在视图类文件中定义一个绘制 Bézier 曲线的宏：

```
#define BEZIERLINE    41
```

然后，在应用程序工具栏或者菜单栏中设置一个绘制 Bézier 曲线的图标标识，通过类向导在视图类中创建该图标标识的消息映射函数，在该函数中设置绘图命令标识变量 m_flag＝BEZIERLINE，并通过 InitOperation()初始化设置绘图窗口。

由于在屏幕上交互拾取控制顶点，所以在单击函数 OnLButtonDown()中拾取点，同时对拾取的点集合调用求值器命令进行曲线点映射。该函数代码如下：

```
void COpenGL001View::OnLButtonDown(UINT nFlags, CPoint point) {
    if(m_Rflag == 1&&(m_flag == BEZIERLINE)){     //拾取点,并转换为绘图点
        GLPoint Pt;
        if(aspect_ratio < 1){
            Pt.x = (point.x - this -> winWidth)/winWidth * Win_Size;
            Pt.y = (this -> winHeight - point.y)/winHeight/aspect_ratio * Win_Size;
            Pt.z = 0.0;
        }
        else {
            Pt.x = (point.x - this -> winWidth)/winWidth * aspect_ratio * Win_Size;
            Pt.y = (this -> winHeight - point.y)/winHeight * Win_Size;
            Pt.z = 0.0;
        }
        m_Point_Array.Add(Pt);
        if(m_flag == BEZIERLINE){
            int Pt_Num = m_Point_Array.GetSize();
            if(Pt_Num > 1) {                          //从拾取点构造控制顶点
                GLPoint pt;
                GLfloat ( * quad)[3];
                quad = new GLfloat[Pt_Num][3];
                for(int i = 0;i < Pt_Num;i++){
                    pt = m_Point_Array.GetAt(i);
                    pt.z = 0.0;
                    quad[i][0] = pt.x;
                    quad[i][1] = pt.y;
                    quad[i][2] = pt.z;
                }
                //调用求值器
                glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,3,Pt_Num,&quad[0][0]);
                delete [ ]quad;
            }
```

```
        }
        Invalidate();
    }
    //(原有代码,此处省略)
}
```

在绘图函数 RenderScene()中加入绘制 Bézier 曲线的代码,使用的是第二种绘制 Bézier 曲面的方法。代码放在最后一行的 glPopMatrix()之前,参考如下:

```
if(m_flag == BEZIERLINE){
    if(m_Point_Array.GetSize()> 1){
        glEnable(GL_MAP1_VERTEX_3);                //启用求值器
        glMapGrid1d(100,0.0,1.0);                  //创建一维网格
        glEvalMesh1(GL_LINE,0,100);                //绘制一维网格,即 Bézier 曲线
    }
}
```

运行程序,单击绘制 Bézier 曲线图标后,在屏幕上拾取点作为曲线的控制顶点,当点集数量超过 2 个后,即在屏幕上绘制一条 Bézier 曲线。

2. 绘制 Bézier 曲面

OpenGL 绘制 Bézier 曲面的方法和绘制 Bézier 曲线的方法相同,第一步也是将绘制的曲面的控制顶点通过求值器映射曲面上点。二维求值器命令的原型为:

```
void glMap2d/f(GLenum target,Type u1,Type u2,GLint ustride,GLint uorder,Type v1,Type v2,GLint
vstride,GLint vorder,const Type * points);
```

该命令的大部分参数和 glMap1d/f()命令中的含义相同。参数 v1、v2 为曲面在 v 向变量的参数值下限和上限。vstride 为 v 向同一 u 参数值在控制顶点数组 points 中相邻点之间的数据间距,假定每个顶点用(x,y,z)表示,曲面在 u 向的控制顶点为 u_num。则 v 向控制点数据的间距为 3 * u_num,参数 points 为绘制曲面的控制顶点网格数组的首地址,当顶点集合是二维数组时,按 u 向的顺序排列顶点,即排列 u 向第一行顶点后,再排列第二行、第三行……。

同样,利用求值器创建控制顶点和曲面上点映射后,必须利用 GL_MAP2_VERTEX_3 作为参数值调用 glEnable()命令,才能启用求值器生成曲面上的点。

当考虑绘制的曲面具有真实感光照效果时,需要为构成曲面的每一个小平面指定法向量。OpenGL 提供了自动法向量生成功能,程序利用 GL_AUTO_NORMAL 作为常量值调用 glEnable()命令即可。

根据求值器创建的曲面上的点绘制 Bézier 曲面时,像绘制 Bézier 曲线一样,调用 glMapGrid2()函数设置一个网格,在参数 u、v 方向上形成二维网格。函数原型为:

```
void glMapGrid2X(GLint un,Type u1,Type u2,GLint vn,Type v1,Type v2);
```

其中,参数 un 和 vn 分别为网格在 u 向和 v 向的分段数,v1、v2 分别为 v 向参数定义域的下限和上限,其他和 glMapGrid1X()中的同名参数意义相同。

然后,调用 glEvalMesh2()函数连接网格各面片,产生 Bézier 曲面。函数原型为:

```
void glEvaMesh2(GLenum mode,GLint i1,GLint i2,GLint j1,GLint j2);
```

其中，参数 mode 指定绘制模式，取值可以为 GL_FILL 或者 GL_LINE；参数 i1、i2 为 u 方向的网格的范围值；j1、j2 为 v 方向的网格范围值。

这样就可以绘制一个 Bézier 曲面。

在程序 OpenGL001 中绘制 Bézier 曲面时，可以将绘制 Bézier 曲线的控制顶点沿着 z 轴方向平移一段距离，获得一组或多组新的控制顶点，利用这几组控制顶点创建 Bézier 曲面。例如，在 z 轴得到一组新控制顶点，则先后两组控制顶点可以创建 v 向二阶的 Bézier 曲面。

首先，在视图类文件中定义一个绘制 Bézier 曲面的宏：

```
#define BEZIERSURF    42
```

然后，在应用程序工具栏或者菜单栏中设置一个绘制 Bézier 曲面的图标标识，通过类向导在视图类中创建该图标标识的消息映射函数，例如为 OnBézierSurf()，在该函数中设置绘图命令标识变量 m_flag＝BEZIERSURF，并创建曲面控制顶点网格和调用求值器命令。代码如下：

```
void COpenGL001View::OnBézierSurf() {
    m_flag = BEZIERSURF;
    glEnable(GL_MAP2_VERTEX_3);                    //启用求值器
    glEnable(GL_AUTO_NORMAL);                      //启用自动法向量生成功能
    int Pt_Num = m_Point_Array.GetSize();
    if(Pt_Num > 1) {//创建曲面控制顶点网格
        GLPoint pt;
        GLfloat ( * quad)[3];
        quad = new GLfloat[Pt_Num * 2][3];
        for(int i = 0;i < Pt_Num;i++){
            pt = m_Point_Array.GetAt(i);
            pt.z = 0.0;
            quad[i][0] = pt.x;
            quad[i][1] = pt.y;
            quad[i][2] = pt.z;
            pt.z = 1.0;
            quad[i + Pt_Num][0] = pt.x;
            quad[i + Pt_Num][1] = pt.y;
            quad[i + Pt_Num][2] = pt.z;
        }
    //调用求值器
    glMap2f(GL_MAP2_VERTEX_3,0,1,3,Pt_Num,0,1,Pt_Num * 3,2, &quad[0][0]);
        delete [ ]quad;
    }
    m_rAngle = 20.0;                               //将坐标系旋转一定角度，以便显示立体感
    Invalidate();
}
```

在绘图函数 RenderScene()中加入绘制 Bézier 曲面的代码，此代码放在最后一行的 glPopMatrix()之前，参考如下：

```
if(m_flag == BEZIERSURF){
    if(m_Point_Array.GetSize()> 1){
        glMapGrid2f(50,0.0,1.0,10,0,1.0);       //设置曲面网格
        glEvalMesh2(GL_LINE,0,50,0,50);         //网格绘制
    }
}
```

运行程序,在绘制 Beizier 曲线后,单击绘制 Bézier 曲面命令,即可实现绘制。将网络绘制函数 glEvalMesh2(GL_LINE,0,50,0,50)中的 GL_LINE 修改为 GL_FILL 常量,曲面由网格线显示改变为实体渲染显示。

10.5.3 绘制 NURBS 曲线曲面

1. 绘制 NURBS 曲线

OpenGL 绘制 NURBS 曲线曲面的功能由 GLU 库提供,当获得了一组绘制 NURBS 曲线的控制顶点后,采用下面的步骤绘制 NURBS 曲线。

第一步:创建一个 NURBS 对象。定义一个 NURBS 对象类型的指针变量,并创建和初始化,代码如下:

```
GLUnurbsObj  * pNurb;
pNurb = gluNewNurbsRenderer();
```

第二步:设置 NURBS 对象的一些属性,对于 NURBS 曲线可以采用默认的属性值。

第三步:绘制 NURBS 曲线。绘制 NURBS 曲线必须在 gluBeginCurve() 和 gluEndCurve()命令对之间进行,这两个命令的原型分别为:

```
void gluBeginCurve(GLUnurbsObj * pNurb);
void gluEndCurve(GLUnurbsObj * pNurb);
```

它们的参数 pNurb 为指向 NURBS 对象的指针。具体的绘制曲线的工作由命令 gluNurbsCurve()来完成,该命令的原型为:

```
void gluNurbsCurve(GLUnurbsObj  * pNurb, GLint nknots, GLfloat  * knot, GLint stride, GLfloat  *
ctrlarray, GLint order, GLenum type);
```

其中,参数 pNurb 为一个指向 NURBS 对象的指针;参数 nknots 为节点数组的长度;knot 为节点数组的首地址;stride 为每个控制顶点在控制顶点数组的间距;ctrlarray 为控制顶点数组的首地址;order 为 NURBS 曲线的阶数;type 为 NURBS 对象的数据类型,值为 GL_MAP1_VERTEX_3 或者 GL_MAP1_COLOR_4。

第四步:删除 NURBS 对象。

当不再需要某个 NURBS 对象时,可以调用 glDeleteNurbsRenderer()命令删除该对象,以将所占用的内存返还给系统,函数原型为:

```
Void glDeleteNurbsRenderer(GLUnurbsObj * pNurb);
```

在程序 OpenGL001 中绘制 NURBS 曲线时,可以像绘制 Bézier 曲线一样,首先通过鼠标在屏幕坐标系拾取控制顶点,然后再绘制曲线。

首先，在视图类文件中定义一个绘制NURBS曲线的宏：

```
#define NURBSLINE    43
```

在视图类COpenGL001View中定义NURBS对象指针：

```
GLUnurbsObj * pNurb;                    //NURBS对象
```

并在COpenGL001View的构造函数COpenGL001View()中创建和初始化NURBS对象：

```
pNurb = gluNewNurbsRenderer();          //创建NURBS对象
```

在COpenGL001View的析构函数COpenGL001View()中删除NURBS对象指针：

```
gluDeleteNurbsRenderer(pNurb);          //删除NURBS对象
```

在应用程序工具栏或者菜单栏中设置一个绘制NURBS曲线的图标标识，通过类向导在视图类中创建该标识的消息映射函数，在该函数中设置绘图命令标识变量为：

```
m_flag = NURBSSPLINE
```

由于NURBS的控制顶点也是通过鼠标在屏幕通过交互拾取获得，因此，在单击函数OnLButtonDown()中，将拾取条件的代码修改为在绘制NURBS曲线时也可以拾取屏幕点，即为：

```
if(m_Rflag == 1&&(m_flag == BEZIERLINE||m_flag == NURBSLINE))
```

在绘图函数RenderScene()中加入绘制NURBS曲线的代码，由于拾取的控制顶点的数量可能多也可能少，所以，绘制的NURBS曲线的次数可以不相同。当拾取两个控制顶点时，绘制二阶NURBS曲线；当拾取三个控制顶点时，绘制三阶NURBS曲线；当拾取四个及以上数量的控制顶点时，均绘制工程上常用的四阶NURBS曲线。节点数组knots在两端重复度为曲线的阶数4，中间重复度均为1。绘制曲线代码放在最后一行的glPopMatrix()之前，参考如下：

```
if(m_flag == NURBSLINE){
    int Pt_Num = m_Point_Array.GetSize();
    if(Pt_Num > 1) {
        //建立控制顶点数组
        GLPoint pt;
        GLfloat ( * ctrlPoints)[3];
        GLfloat * Knots;                //节点数组
        ctrlPoints = new GLfloat[Pt_Num][3];
        for(int i = 0;i < Pt_Num;i++){
            pt = m_Point_Array.GetAt(i);
            pt.z = 0.0;
            ctrlPoints[i][0] = pt.x;
            ctrlPoints[i][1] = pt.y;
            ctrlPoints[i][2] = pt.z;
        }
        //建立节点数组
```

```
            if(Pt_Num == 2){                        //一次 NURBS
                Knots = new GLfloat[Pt_Num + 2];
                for(i = 0;i < Pt_Num + 2;i++){
                    if(i < 2)
                        Knots[i] = 0;
                    else
                        Knots[i] = 1;
                }
                //绘制二阶(一次)NURBS 曲线
                gluBeginCurve(pNurb);
                gluNurbsCurve(pNurb,4,Knots,3,&ctrlPoints[0][0],2,GL_MAP1_VERTEX_3);
                gluEndCurve(pNurb);
                delete []Knots;
            }
            if(Pt_Num == 3){                        //二次 NURBS
                Knots = new GLfloat[Pt_Num + 3];
                //利用准均匀节点数组两端节点重复度 3,中间重复度 1
                for(i = 0;i < Pt_Num + 3;i++){
                    if(i < 3)
                        Knots[i] = 0;
                    else if(i >= 3&&i <= Pt_Num)
                        Knots[i] = i - 2;
                    else
                        Knots[i] = Pt_Num - 2;
                }
                //绘制三阶(二次)NURBS 曲线
                gluBeginCurve(pNurb);
                gluNurbsCurve(pNurb,Pt_Num + 3,Knots,3,&ctrlPoints[0][0],3,GL_MAP1_VERTEX_3);
                gluEndCurve(pNurb);
                delete []Knots;
            }
            if(Pt_Num > 3){                         //三次 NURBS
                Knots = new GLfloat[Pt_Num + 4];
                //利用准均匀节点数组两端节点重复度 4,中间重复度 1
                for(i = 0;i < Pt_Num + 4;i++){
                    if(i < 4)
                        Knots[i] = 0;
                    else if(i >= 4&&i <= Pt_Num)
                        Knots[i] = i - 3;
                    else
                        Knots[i] = Pt_Num - 3;
                }
                //绘制四阶(三次)NURBS 曲线
                gluBeginCurve(pNurb);
                gluNurbsCurve(pNurb,Pt_Num + 4,Knots,3,&ctrlPoints[0][0],4,GL_MAP1_VERTEX_3);
                    gluEndCurve(pNurb);
                delete []Knots;
            }
            delete []ctrlPoints;
        }
    }
```

执行应用程序，单击绘制 NURBS 曲线命令，在屏幕上拾取点，可以实时绘制一条通过首末拾取点的 NURBS 曲线。

2. 绘制 NURBS 曲面

OpenGL 绘制 NURBS 曲面的步骤和绘制 NURBS 曲线的相同，也需要创建 NURBS 对象等。对于曲面绘制，在绘制前非常有必要设置 NURBS 对象的属性。设置 NURBS 对象属性需要调用 GLU 库中的 gluNurbsProperty()命令，该命令的原型为：

```
void gluNurbsProperty(GLUnurbsObj *nobj,GLenum property,GLfloat value);
```

其中，参数 nobj 为指向 NURBS 对象的指针；property 为要设置的属性，该参数值可如表 10.5-2 所示；参数 value 用于给出所设置属性的值，该参数可以是一个数值，也可以是表 10.5-3 中的一个枚举常量。

表 10.5-2　NURBS 属性

枚举常量	意　义
GLU_SAMPLING_TOLERANCE	采样容差(以像素计)，默认值为 50.0
GLU_DISPLAY_MODE	NURBS 曲面的渲染方法
GLU_CULLING	指出是否使用镶嵌
GLU_AUTO_LOAD_MATRIX	指出是否从服务器下载投影、模型视图矩阵及视口
GLU_PARAMETRIC_TOLERANCE	最大采样距离(以像素计)，默认值为 0.5
GLU_SAMPLING_METHOD	NURBS 曲面的镶嵌方法
GLU_U_STEP	u 方向每个单位长度上的采样点数
GLU_V_STEP	v 方向每个单位长度上的采样点数

表 10.5-3　value 参数值

枚举常量	意　义
GLU_FILL	曲面以多边形形式渲染
GLU_OUTLINE_POLYGON	仅绘制多边形外框
GLU_OUTLINE_PATCH	仅绘制用户定义的片段和修剪曲线的外框
GLU_PATH_LENGTH	指出多边形边缘的最大长度，所渲染的表面不大于 GLU_SAMPLING_TOLERANCE 所指定的值
GLU_PARAMETRIC_ERROR	指出表面利用 GLU_PARAMETRIC_TOLERANCE 所指定值渲染
GLU_DOMAIN_DISTANCE	指定 u 方向和 v 方向的单位采样点个数(以参数坐标计)

绘制 NURBS 曲面的函数需要放在 gluBeginSurface()和 gluEndSurface()两个命令之间进行，这两个函数的原型为：

```
void gluBeginSurface(GLUnurbsObj *nobj);
void gluEndSurface(GLUnurbsObj *nobj);
```

其中，参数 nobj 为 NURBS 对象的指针。

绘制 NURBS 曲面的函数为 gluNurbsSurface()，该函数原型为：

```
void gluNurbsSurface(GLUnurbsObj *nobj,GLint sknot_count,GLfloat *sknot,GLint tknot_count,
GLfloat *tknot,GLint s_stride,GLint t_stride,GLfloat *ctrlarray,GLint sorder,GLint torder,
GLenum type);
```

其中，参数 nobj 为 NURBS 对象的指针；参数 sknot_count 和 tknot_count 分别为 u 向和 v 向的节点数组长度；参数 sknot 和 tknot 分别为 u 向和 v 向的节点数组；参数 ctrlarray 为曲面的控制顶点网格数组；参数 s_stride 和 t_stride 分别为控制顶点数组 ctrlarray 在 u 向和 v 向的对应数据间距，假设在控制顶点以(x,y,z)表示，在 u 向的控制顶点数为 num，则 s_stride 值为 3，t_stride 值为 num * 3；sorder 和 torder 分别为曲面在 u 向和 v 向的阶数；参数 type 为曲面类型，其取值为 GL_MAP2_VERTEX_3 或者 GL_MAP2_COLOR_4。

在程序 OpenGL001 中绘制 NURBS 曲面时，可以在已绘制的 NURBS 曲线的基础上绘制 NURBS 曲面。为了获得 NURBS 曲面的控制顶点网格，将绘制 NURBS 曲线的控制顶点沿着 z 轴方向平移一段距离，获得一组或多组新的控制顶点，利用这几组控制顶点形成的控制顶点网格创建 NURBS 曲面。例如，将控制顶点在 z 轴方向平移三次得到三组新的控制顶点，这样加上原有的控制顶点组，在 v 向有四组控制顶点，就可以创建 v 向四阶的 NURBS 曲面。

首先，在视图类文件中定义一个绘制 NURBS 曲面的宏：

```
#define NURBSSURF     44
```

在视图类 COpenGL001View 中定义 NURBS 对象指针：

```
GLUnurbsObj  * pNurb_Surf;                        //NURBS 对象_曲面
```

并在 COpenGL001View 的构造函数 COpenGL001View()中创建和初始化 NURBS 对象并同时设置对应 NURBS 对象的相关属性：

```
pNurb_Surf = gluNewNurbsRenderer();                              //创建 NURBS 对象
gluNurbsProperty(pNurb_Surf,GLU_SAMPLING_TOLERANCE,20.f);  //设置属性
gluNurbsProperty(pNurb_Surf,GLU_DISPLAY_MODE,GLfloat(GLU_FILL));
```

在 COpenGL001View 的析构函数 COpenGL001View()中删除 NURBS 对象指针：

```
gluDeleteNurbsRenderer(pNurb_Surf);                   //删除 NURBS 对象
```

然后，在应用程序工具栏或者菜单栏中设置一个绘制 NURBS 曲面的图标标识，通过类向导在视图类中创建该标识的消息映射函数，例如为 OnNurbsSurf()，在该函数中设置绘图命令标识变量及启用自动法向量计算，代码如下：

```
void COpenGL001View::OnNurbsSurf() {
    m_flag = NURBSSURF;
    glEnable(GL_AUTO_NORMAL);
    Invalidate();
}
m_flag = NURBSSURF;
glEnable(GL_AUTO_NORMAL);
```

在绘图函数 RenderScene()中加入绘制 NURBS 曲面的代码，曲面在 u 向的阶数和绘制的 NURBS 曲线的阶数的设置方法相同，v 向也为四阶，v 向节点数组 tknots 在两端重复度为 v 向的阶数 4，中间重复度均为 1。代码放在最后一行的 glPopMatrix()之前，参考如下：

```
if(m_flag == NURBSSURF){
    int Pt_Num = m_Point_Array.GetSize();
    if(Pt_Num > 1) {
        //建立控制顶点数组
        GLPoint pt;
        GLfloat ( * ctrlPoints)[3];                  //控制顶点数组
        GLfloat  * Knots;                            //节点数组
        GLfloat Knott[8] = {0.0f,0.0f,0.0f,0.0f,1.0f,1.0f,1.0f,1.0f};       //v 向节点数组
        ctrlPoints = new GLfloat[Pt_Num * 4][3];     //动态设置控制顶点数组长度
        for(int i = 0;i < Pt_Num;i++){
            pt = m_Point_Array.GetAt(i);
            pt.z = 0.0;
            ctrlPoints[i][0] = pt.x;
            ctrlPoints[i][1] = pt.y;
            ctrlPoints[i][2] = pt.z;
            pt.z = 0.33 * Win_Size;                  //沿 z 轴平移一个距离,获得新控制顶点
            ctrlPoints[i + Pt_Num][0] = pt.x;
            ctrlPoints[i + Pt_Num][1] = pt.y;
            ctrlPoints[i + Pt_Num][2] = pt.z;
            pt.z = 0.65 * Win_Size;                  //沿 z 轴平移一个距离,获得新控制顶点
            ctrlPoints[i + Pt_Num * 2][0] = pt.x;
            ctrlPoints[i + Pt_Num * 2][1] = pt.y;
            ctrlPoints[i + Pt_Num * 2][2] = pt.z;
            pt.z = 1.0 * Win_Size;                   //沿 z 轴平移一个距离,获得新控制顶点
            ctrlPoints[i + Pt_Num * 3][0] = pt.x;
            ctrlPoints[i + Pt_Num * 3][1] = pt.y;
            ctrlPoints[i + Pt_Num * 3][2] = pt.z;
        }
        if(Pt_Num == 2){                             //一次 NURBS
            Knots = new GLfloat[Pt_Num + 2];
            for(i = 0;i < Pt_Num + 2;i++){           //建立 u 向节点数组
                if(i < 2)
                    Knots[i] = 0;
                else
                    Knots[i] = 1;
            }
            gluBeginSurface(pNurb_Surf);
                gluNurbsSurface(pNurb_Surf,4,Knots,8,Knott,3,Pt_Num * 3,&ctrlPoi nts[0][0],2,
4,GL_MAP2_VERTEX_3);
            gluEndSurface(pNurb_Surf);
            delete [ ]Knots;
        }
        if(Pt_Num == 3){                             //二次 NURBS
            Knots = new GLfloat[Pt_Num + 3];
            //利用准均匀节点向量两端节点重复度 3,中间重复度 1
            for(i = 0;i < Pt_Num + 3;i++){           //建立 u 向节点数组
                if(i < 3)
                    Knots[i] = 0;
                else if(i >= 3&&i < Pt_Num)
                    Knots[i] = i - 2;
                else
```

```
                    Knots[i] = Pt_Num - 2;
                }
                gluBeginSurface(pNurb_Surf);
                    gluNurbsSurface(pNurb_Surf,6,Knots,8,Knott,3,Pt_Num * 3,&ctrlPoi nts[0][0],3,
4,GL_MAP2_VERTEX_3);
                gluEndSurface(pNurb_Surf);
                delete []Knots;
            }
            if(Pt_Num>3){//三次 NURBS
                Knots = new GLfloat[Pt_Num + 4];
                //利用准均匀节点向量两端节点重复度 m + 1,中间重复度小于 m
                for(i = 0;i < Pt_Num + 4;i++){
                    if(i < 4)
                        Knots[i] = 0;
                    else if(i >= 4&&i < Pt_Num)
                        Knots[i] = i - 3;
                    else
                        Knots[i] = Pt_Num - 3;
                }
                gluBeginSurface(pNurb_Surf);
                    gluNurbsSurface(pNurb_Surf, Pt_Num + 4, Knots, 8, Knott, 3, Pt_Num * 3,
&ctrlPoints[0][0],4,4,GL_MAP2_VERTEX_3);
                gluEndSurface(pNurb_Surf);
                delete []Knots;
            }
            delete []ctrlPoints;
        }
}
```

运行程序,首先单击绘制 NURBS 曲线命令,在屏幕拾取控制顶点并绘制曲线,然后再单击绘制 NURBS 曲面的命令,根据上述代码即可绘制 NURBS 曲面。图 10.5-1 所示为绘制的一个 NURBS 曲面的效果图。

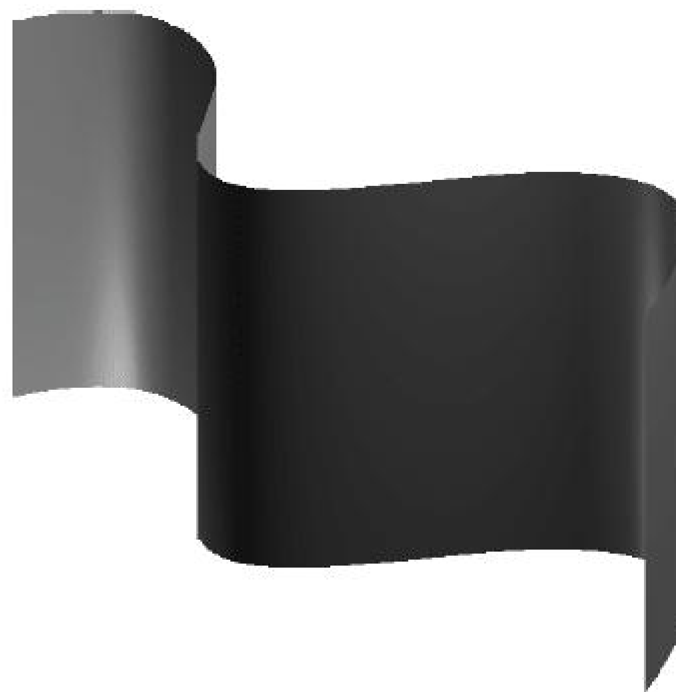


图 10.5-1 绘制 NURBS 曲面

10.5.4 NURBS 曲面修剪

OpenGL 允许对 NURBS 曲面进行修剪。修剪的意思是在 NURBS 表面上创建删除部分,例如在表面上开一些孔,或者将曲面的某些边角剪掉。

修剪曲面除了必须有曲面外,还必须有一个封闭无交叉的修剪曲线。可以创建两种修剪曲线:分段的线性曲线和 NURBS 曲线。前者需要调用 gluPwlCurve()命令来创建,后者必须通过前文已经介绍过的 gluNurbsCurve()命令(用于修剪时,其中的参数 type 取值为 GLU_MAP1_TRIM_2)来创建。gluPwlCurve()命令的原型为:

```
void gluPwlCurve(GLUnurbsObj * nobj,GLint count,GLfloat * array,GLint stride,GLenum type);
```

其中,参数 nobj 为一个指向 NURBS 对象的指针;参数 count 为构成修剪曲线的顶点数;array 为顶点数组的首地址,由于要求修剪曲线是封闭的,那么,当用一个线性曲线形成

封闭形状时，修剪曲线顶点数组首末位的顶点必须是相同的；stride 为顶点数组中的对应坐标位的数据间距；type 为曲线类型，其值一般取 GLU_MAP1_TRIM_2 或 GLU_MAP1_TRIM_3(较少使用)。

对于修剪曲线围成的修剪窗口有两点需要注意：第一是两种创建修剪曲线命令的顶点是曲面参数空间(u、v)定义域的值，而非曲面上点的坐标值；第二，修剪窗口外环的顶点顺序在参数空间(u、v)内按逆时针走向，内环按顺时针走向。

NURBS 曲线修剪的操作步骤与绘制 NURBS 曲面基本相同，但是修剪操作必须放在调用绘制曲面的 gluBeginSurface()和 gluNurbsSurface()命令之后，然后调用 gluBeginTrim()命令开始修剪，创建修剪曲线，再调用 gluEndTrim()命令终止修剪。这两个命令原型为：

```
gluBeginTrim(GLUnurbsObj * nobj);
gluEndTrim(GLUnurbsObj * nobj);
```

其中，参数 nobj 为一个指向 NURBS 对象的指针。

在程序 OpenGL001 中进行 NURBS 曲面修剪时，可以在已绘制的 NURBS 曲面的基础上进行。在曲面的参数域(u,v)内构造一个外环线性曲线和一个内环线性曲线。

首先，在 COpenGL001View 类中设置一个 NURBS 曲面修剪的标识变量：

```
int NurbsTrimflag;
```

然后在应用程序工具栏或者菜单栏中设置一个 NURBS 曲面修剪的图标标识，通过类向导在视图类中创建该标识的消息映射函数，例如为 OnNurbsTrim()，在该函数中设置修剪指令，代码如下：

```
void COpenGL001View::OnNurbsTrim() {
    NurbsTrimflag = 1;                          //修剪
    Invalidate();
}
```

在绘图函数 RenderScene()中，在绘制 NURBS 曲面的部分加入曲面修剪的代码，相关代码如下：

```
if(m_flag == NURBSSURF){
    int Pt_Num = m_Point_Array.GetSize();
    if(Pt_Num > 1){
        ...//建立曲面控制顶点数组和节点数组代码部分，前文已列出，本处省略
        GLfloat trimPoints_out[5][2];           //修剪曲线顶点外环
        GLfloat trimPoints_in[5][2];            //修剪曲线顶点内环
        if(NurbsTrimflag == 1){                 //修剪，则构造修剪曲线内外环
            trimPoints_out[0][0] = trimPoints_out[4][0] = 0.1;
            trimPoints_out[0][1] = trimPoints_out[4][1] = 0.1;
            trimPoints_out[1][0] = 0.9;
            trimPoints_out[1][1] = 0.1;
            trimPoints_out[2][0] = 0.9;
            trimPoints_out[2][1] = 0.9;
            trimPoints_out[3][0] = 0.1;
            trimPoints_out[3][1] = 0.9;
            trimPoints_in[0][0] = trimPoints_in[4][0] = 0.35;
```

```
            trimPoints_in[0][1] = trimPoints_in[4][1] = 0.35;
            trimPoints_in[1][0] = 0.35;
            trimPoints_in[1][1] = 0.70;
            trimPoints_in[2][0] = 0.70;
            trimPoints_in[2][1] = 0.70;
            trimPoints_in[3][0] = 0.70;
            trimPoints_in[3][1] = 0.35;
        }
        if(Pt_Num == 2){                              //一次 NURBS
            ...//建立曲面控制顶点数组和节点数组代码部分,前文已列出,本处省略
            gluBeginSurface(pNurb_Surf);
            gluNurbsSurface(pNurb_Surf,4,Knots,8,Knott,3,Pt_Num * 3,&ctrlPoints[0][0],2,
4,GL_MAP2_VERTEX_3);
                if(NurbsTrimflag == 1){               //修剪
                    gluBeginTrim(pNurb_Surf);
                        gluPwlCurve(pNurb_Surf,5,&trimPoints_out[0][0],2,GLU_MAP1_TRIM_2);
                    gluEndTrim(pNurb_Surf);
                    gluBeginTrim(pNurb_Surf)
                        gluPwlCurve(pNurb_Surf,5,&trimPoints_in[0][0],2,GLU_MAP1_TRIM_2);
                    gluEndTrim(pNurb_Surf);
                }
            gluEndSurface(pNurb_Surf);
            delete [ ]Knots;
        }
        if(Pt_Num == 3){                              //二次 NURBS
            ...//建立曲面控制顶点数组和节点数组代码部分,前文已列出,本处省略
            gluBeginSurface(pNurb_Surf);
                gluNurbsSurface(pNurb_Surf,6,Knots,8,Knott,3,Pt_Num * 3,&ctrlPoints[0][0],3,4,
GL_MAP2_VERTEX_3);
                if(NurbsTrimflag == 1){               //修剪
                    gluBeginTrim(pNurb_Surf);
                        gluPwlCurve(pNurb_Surf,5,&trimPoints_out[0][0],2,GLU_MAP1_TRIM_2);
                      gluEndTrim(pNurb_Surf);
gluBeginTrim(pNurb_Surf)
gluPwlCurve(pNurb_Surf,5,&trimPoints_in[0][0],2,GLU_MAP1_TRIM_2);
                    gluEndTrim(pNurb_Surf);
                }
            gluEndSurface(pNurb_Surf);
            delete [ ]Knots;
        }
        if(Pt_Num > 3){                               //三次 NURBS
            ...//建立节点数组代码部分,前文已列出,本处省略
            gluBeginSurface(pNurb_Surf);
                  gluNurbsSurface(pNurb_Surf, Pt_Num + 4, Knots, 8, Knott, 3, Pt_Num * 3,
&ctrlPoints[0][0],4,4,GL_MAP2_VERTEX_3);
                if(NurbsTrimflag == 1){               //修剪
                    gluBeginTrim(pNurb_Surf);
                        gluPwlCurve(pNurb_Surf,5,&trimPoints_out[0][0],2,GLU_MAP1_TRIM_2);
                    gluEndTrim(pNurb_Surf);
                    gluBeginTrim(pNurb_Surf);
                        gluPwlCurve(pNurb_Surf,5,&trimPoints_in[0][0],2,GLU_MAP1_TRIM_2);
```

```
                    gluEndTrim(pNurb_Surf);
                }
            gluEndSurface(pNurb_Surf);
            delete [ ]Knots;
        }
        delete [ ]ctrlPoints;
    }
}
```

执行程序，绘制 NURBS 曲面后单击 NURBS 修剪命令，则绘制的 NURBS 曲面被裁剪变小，中间有个方孔。

10.5.5　曲面纹理映射

纹理图像不仅可以映射到平面上，也可以映射到任意曲面上，包括 Bézier 曲面和 NURBS 曲面。在曲面上实现纹理映射的步骤和前文的平面纹理映射的步骤相同，即创建和绑定纹理对象、加载纹理图像、设置纹理参数、设置环境参数，建立纹理坐标和曲面上点之间的对应关系，启用纹理坐标映射，实现在曲面上映射纹理图像。

对于曲面纹理映射，需要生成和曲面上点对应的纹理坐标。由于 GLU 库根据控制顶点和节点直接生成了曲面，没有机会指定纹理坐标，因此，不能直接建立曲面点和纹理坐标之间的映射关系。但是，可以在纹理图像的参数(s,t)和曲面的参数(u,v)之间建立对应关系，通过参数之间的一一对应实现纹理图像到曲面的映射。

一般情况下，曲面和纹理的参数域均定义在(0,0)、(1,0)、(1,1)、(0,1)范围内，通过调用求值器命令 glMap2f()，在纹理坐标的参数和曲面的参数建立对应关系，在 glMap2f()命令中的数值类型取纹理坐标映射类型：GL_MAP2_TEXTURE_COORD_2，这样，产生的是纹理坐标的参数值。

然后再利用 GL_MAP2_TEXTURE_COORD_2 为参数调用 glEnable()命令，启用纹理坐标参数映射，这时绘制曲面即可实现曲面上的纹理映射。

在程序 OpenGL001 中实现曲面纹理映射时，可以对绘制的 Bézier 曲面或者 NURBS 曲面进行纹理映射。下面设置对绘制的 NURBS 曲面进行纹理映射。

首先，在 COpenGL001View 类中设置一个曲面纹理加载的指令变量和纹理对象的变量：

```
int Surf_Texture_flag;                              //曲面纹理加载的变量
GLuint Surf_texture;                                //纹理对象变量
```

然后，在应用程序工具栏或者菜单栏中设置一个曲面纹理加载的图标标识，通过类向导在视图类中创建该标识的消息映射函数，例如为 OnNurbsSurfTextureMap()，在该函数中设置纹理映射指令，创建纹理对象，加载纹理图像，并进行纹理设置等。代码如下：

```
void COpenGL001View::OnNurbsSurfTextureMap() {      //打开一个图像文件
CFileDialog hFileDlg(true,NULL,NULL,OFN_FILEMUSTEXIST|OFN_ READONLY |OFN_PATHMUSTEXIST,TEXT
("bmp 文件( * .bmp)| * .bmp|"), NULL);
    if(hFileDlg.DoModal() == IDOK) {
```

```
            AUX_RGBImageRec * m_image;
            glPixelStorei(GL_UNPACK_ALIGNMENT,1);
            m_image = auxDIBImageLoad(hFileDlg.GetPathName());
            m_iWidth = m_image -> sizeX;
            m_iHeight = m_image -> sizeY;
            m_pImage = m_image -> data;
            glGenTextures(1,&Surf_texture);              //创建纹理对象
            glBindTexture(GL_TEXTURE_2D,Surf_texture);  //绑定纹理对象
    glTexImage2D(GL_TEXTURE_2D,0,3,m_iWidth,m_iHeight,0,GL_RGB,GL_UNSIGNED_BYTE,m_pImage);
                                                        //加载纹理
        //设置纹理参数和环境参数
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
        glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);
            glEnable(GL_TEXTURE_2D);                    //启用纹理
            Surf_Texture_flag = 1;                      //设置纹理映射指令
            Invalidate();
        }
    }
```

在绘图函数 RenderScene()中,在绘制 NURBS 曲面的部分加入曲面纹理映射的代码。该段代码放在曲面绘制之前的位置,参考如下:

```
    GLfloat texpts[2][2][2] = {{{0,0},{0,1}},{{1.0,0.0},{1.0,1.0}}};
    if(Surf_Texture_flag == 1){
        glBindTexture(GL_TEXTURE_2D,Surf_texture);      //绑定纹理对象
        //调用求值器
        glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1,2,2,0,1,4,2,&texpts[0][0][0]);
        glEnable(GL_MAP2_TEXTURE_COORD_2);              //启用纹理坐标映射
    }
```

执行程序,在绘制出一个 NURBS 曲面后,单击纹理图像加载命令,选择一个图像文件,则程序将图像映射到绘制的 NURBS 曲面上,效果如图 10.5-2 所示。



图 10.5-2 曲面映射

第11章 Web图形开发技术

传统的计算机图形程序开发的思路和方法，首先需要在计算机上安装一个图形开发环境，如 Visual Studio 等，图形程序开发完成后，如在其他计算机上使用，也需要进行安装，这些操作耗时耗力、占用大量存储并且传播性较差。近年来，随着互联网技术的广泛应用，开发具有跨平台、便携性和开放性等特点的 Web 应用系统已成为了程序开发模式的新趋势。Web 环境下的图形应用开发也引起了广泛关注，而在技术方面，用于 Web 页面显示的超文本标记语言(HTML)推出的新标准 HTML5 提供了一些新特性，如用于绘图功能的 canvas 元素等，可以很好地支持图形程序的开发。

JavaScript 是 Web 应用开发中实现网页动态交互的主要语言，开发 Web 图形功能时，所有的图形算法实现、交互操作处理以及数据存储等均通过 JavaScript 来实现。JavaScript 和 Web 应用一样，在前端网页部分也具有开放性的特点，通过调用其他第三方写的成熟的 JavaScript 文件，可以使 Web 应用系统具有非常强大的功能，例如，在 Web 图形开发时调用实现 Web 图形绘制的 JavaScript 文件(如：three.js)，可以在网页上获得类似 OpenGL 实现的图形功能。

本章利用 HTML5 的 canvas 绘图特性和 JavaScript 语言，对 Web 环境下的计算机图形学的基本原理和算法进行了编程实践，通过代码实现，探讨了 Web 图形的开发方法。

11.1 Web 绘图技术的结构概述

11.1.1 HTML 网页文档结构

HTML 全称为 Hypertext Markup Language，译为超文本标记语言。首先要明确一个概念，HTML 不是一种编程语言，而是一种描述性的标记语言，用于描述超文本中内容的显示方式。比如在网页中如何定义一个标题、一段文字、插入一个链接等，都是利用 HTML 标记完成的。其最基本的语法为：<标记>内容</标记>。标记通常都是成对使用的，有一个开头标记就对应一个结束标记。当浏览器从服务器接收到 HTML 文件后，就会解释里面的标记符，然后把标记符相对应的功能表达出来。

HTML 文档一般包含头部区域和主体区域两部分，基本结构由 3 个标签负责组织，分别为：< html >、< head >和< body >。其中< html >标签标识 HTML 文档，< head >标签标识头部区域，< body >标签标识主体区域。一个网页的完成有时候会需要很多的 HTML 标

签，常用的标签有< h2 >、< br >、< load >、< canvas >、< form >、< div >等。

HTML 文件的扩展名为 htm 或 html。可以进行网页制作的软件有很多，目前比较流行的是 Dreamweaver，它将可视布局工具、应用程序开发功能和代码编辑支持组合在一起，使得各个层次的开发人员和设计人员都能够快速创建页面精美的、基于标准的网站和应用程序。而在 Windows 系统中，最简单的文本编辑软件就是文本文档(或记事本)。现在通过基本标签来创建一个基本的 HTML 文件。HTML 文件的创建方法十分简单，在桌面上右击，从弹出的快捷菜单中选择“新建”→“文本文档”命令，在打开的文本文档窗口输入如下代码。此时文本文档的窗口如图 11.1-1 所示。

```
<html>
    <head>
        <meta charset = "utf - 8">
        <title>第一个 HTML 文件</title>
    </head>
    <body>
        <p>
            WebGL 环境下的图形学的开发
        </p>
    </body>
</html>
```

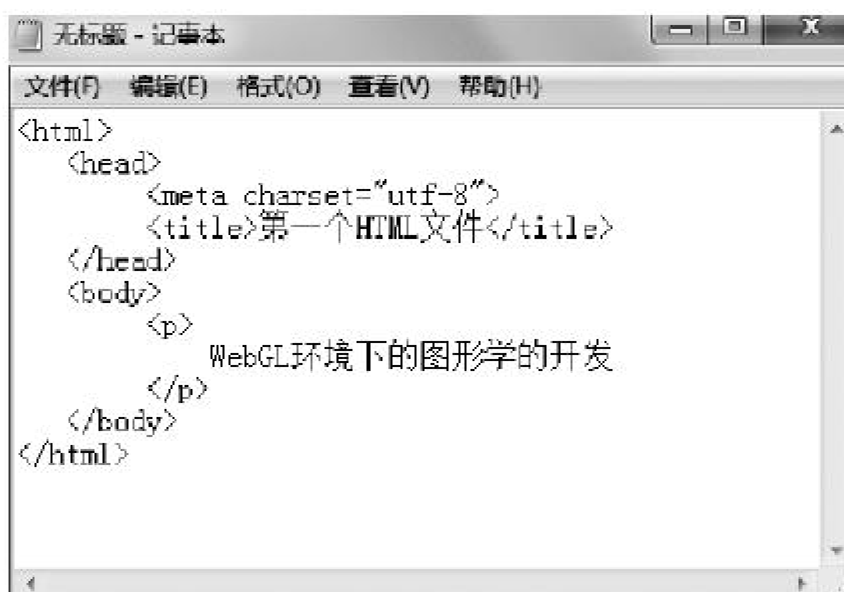


图 11.1-1　在文本文档中输入 HTML 文件内容

编写后保存文档。直接保存便是 txt 格式的普通文档，这样浏览器不会把这个文件当作网页文件对待。选择“另存为”命令，便会弹出如图 11.1-2 所示的对话框。需要注意的是，在“保存类型”下拉列表框中选择“所有文件”选项，然后再在“文件名”文本框中输入一个文件名，并以 htm 或 html 作为文件名后缀。



图 11.1-2　修改后缀名

设置完成后单击“保存”按钮，就会生成 HTML 文件，在 Windows 中，可以看到它的图标就是网页文件图标了，如图 11.1-3 所示。双击图标，就会打开浏览器，显示该文件内容，网页效果如图 11.1-4 所示。

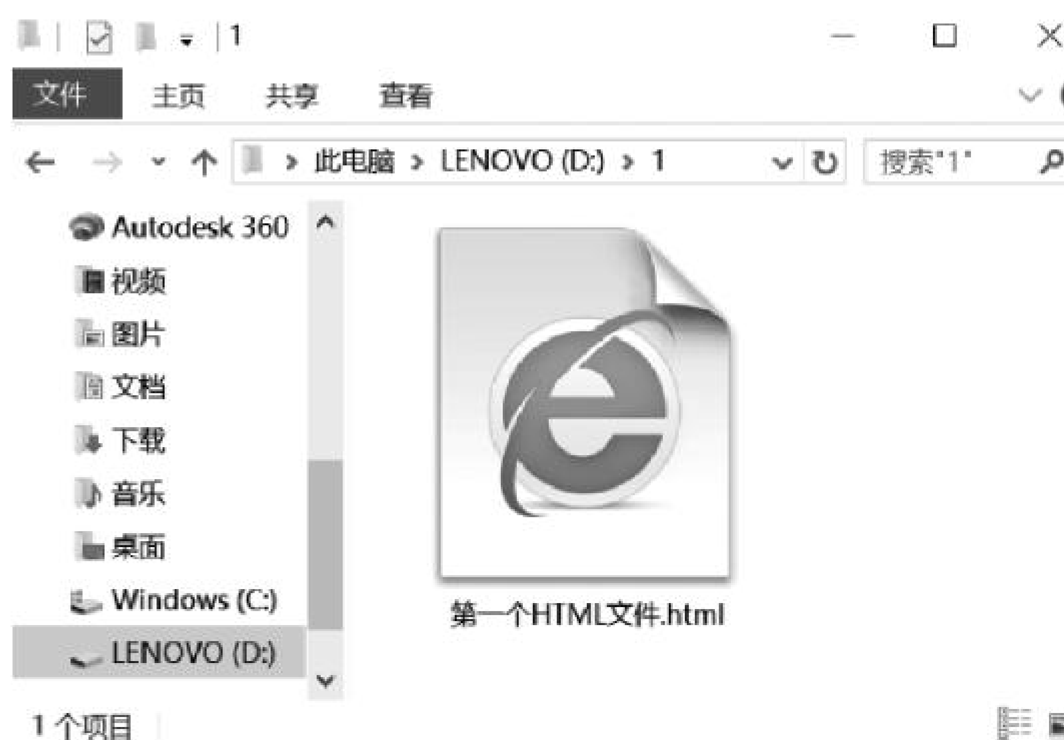


图 11.1-3 HTML 文件

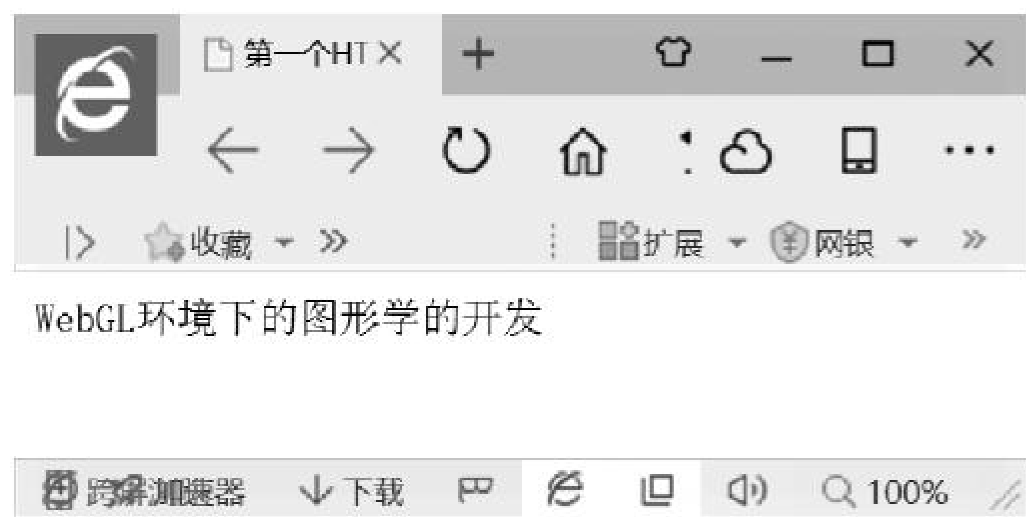


图 11.1-4 在浏览器中的显示效果

在这个 HTML 中，用到了 5 个能够构成完整的最简单的 HTML 文件结构的标记。现在简单介绍这些标记的作用。

<html>放在 HTML 文件的开头，并没有什么实质性的功能，但是在 HTML 文件的开头必须用<html>标记来做一个形式上的开始。

<head>为头标记，紧跟在<html>标记后。<head>中的元素可以引用脚本、指示浏览器在哪里找到样式表、提供元信息等。

<meta>为编码格式声明，否则会出现乱码。

<title>为标题标记，包含在<head>标记内，它的作用是设置网页的标题。在图 11.1-4 中可以看到，网页标题显示在浏览器左上角。

<body>为主体标记，是 HTML 文件的主体部分，网页中最终显示的内容，如段落、表格、超链接、图片等都在这个标记里面。

<p>标识段落文本。

这样第一个基本的 HTML 文件就完成了。相信通过这个简单的例子，大家已经对网页编程有了初步的了解。

11.1.2 JavaScript 概述

JavaScript 是一种基于对象和事件驱动并具有安全驱动的脚本语言，是一种面向 Web 的编程语言，常用来给网页添加各式各样的动态功能。现在绝大多数网站都使用

JavaScript 语言,并且所有的现代 Web 浏览器均包含 JavaScript 解释器,这使得 JavaScript 成为现在使用最广泛的编程语言。通常 JavaScript 脚本嵌入在 HTML 中来实现自身的功能,实现网页交互作用,从而可以开发客户端的应用程序。

现通过一个实例编写一个 JavaScript 程序,通过这个程序说明 JavaScript 的脚本是如何嵌入到 HTML 文档中的。新建文本文档,在里面输入如下所示的代码并保存,网页显示效果如图 11.1-5 所示。

```
<html>
<head>
<meta charset = "utf - 8">
<title>JavaScript 实例</title>
<script>
alert("WebGL 环境下的图形学的开发");
</script></head>
<body></body>
</html>
```

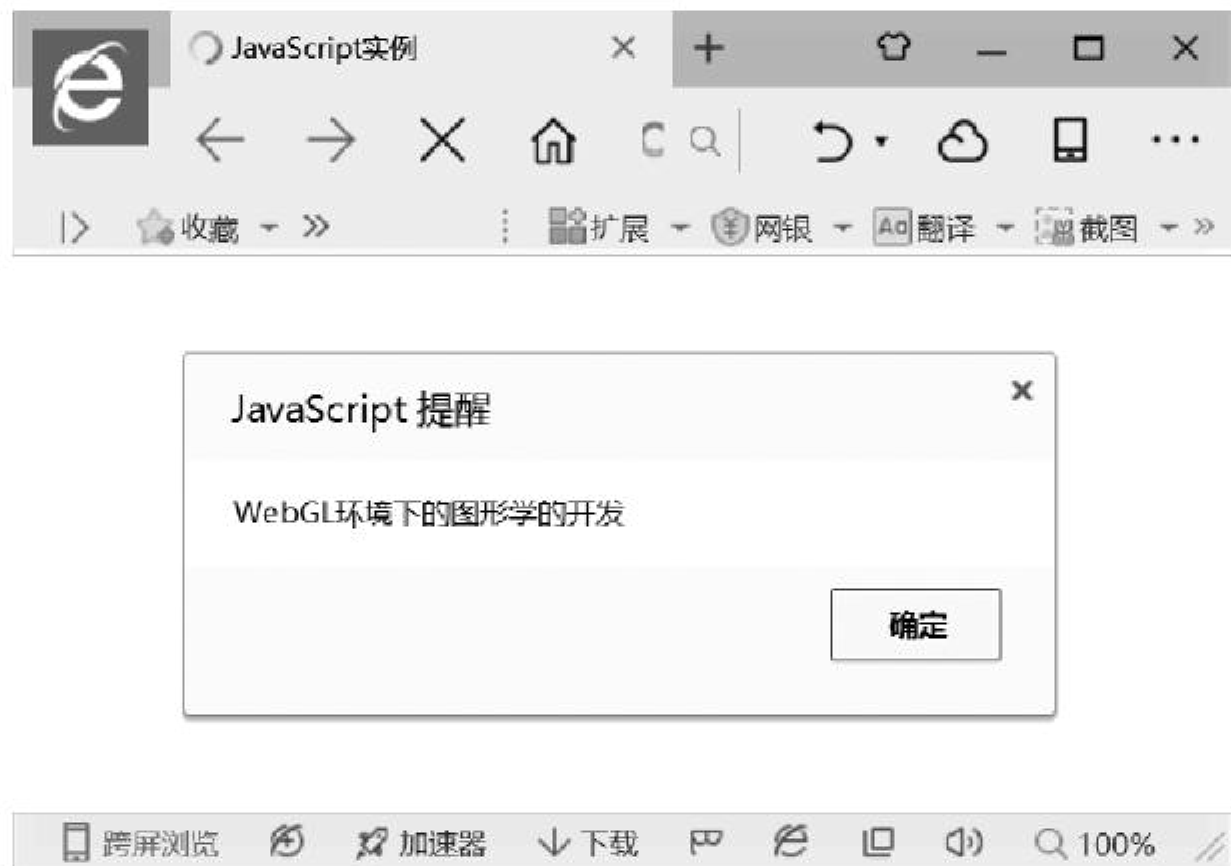


图 11.1-5 JavaScript 实例

JavaScript 代码由<script>说明。在标识<script></script>之间可插入 JavaScript 脚本。

alert()是 JavaScript 的窗口对象方法,是用于显示带有一条指定消息和一个 Ok 按钮的警告框。

这样,通过一个简单的实例可以看出,理解并编写一个 JavaScript 程序并不是十分困难。

11.1.3 canvas 中的图形

canvas 是 HTML 5 中新增的专门用来实现绘图功能的标签,这为 Web 图形的处理打开了广阔的空间。借助 HTML 5 中的<canvas>标签,用户可以在 Web 中绘制各式各样的图形。canvas 的作用是在页面中添加一块矩形画布,在定义的区域中进行绘图操作。其实,canvas 本身功能非常简单,只具有 width 和 height 两种属性,来定义一块无色透明的透明区域。canvas 本身不具有绘图功能,它的功能需要利用 JavaScript 编写绘制在画布中的图形脚本实现。

canvas 在网页中实现画图功能环境的主要代码如下：

```
<html>
<head>
<meta charset = "utf - 8">
<title>用 canvas 实现画布功能</title>
</head>
<body>
<canvas id = "huabu" width = "150" height = "100" style = "border:1px soild;"></canvas>
<script type = "text/javascript">
var c = document.getElementById("huabu");
var context = c.getContext("2d");
</script>
</body>
</html>
```

现在对其中部分代码进行解释。

```
<canvas id = "huabu" width = "150" height = "100" style = "border:1px soild;"></canvas>
```

通过定义 canvas 的 width 和 height 属性，形成一块矩形画布，并且必须定义 canvas 元素的 id 属性，以便于后面的调用；因为画布本身形成的绘图区域不可见，这里为画布添加了线宽为 1 像素的实心边框。

```
var c = document.getElementById("huabu");
```

通过使用 JavaScript 中的 document.getElementById 方法，并通过查找 Id 来寻找 canvas 元素构成的画布的位置。

```
var context = c.getContext("2d");
```

通过使用 canvas 元素的 getContext 方法来获取上下文，在画布中创建 canvas 元素支持的能够绘制图形的 2D 环境。需要注意的是，现在 canvas 元素构成的绘图环境只能支持二维绘图。

最后，通过 canvas 元素形成的绘图区域如图 11.1-6 所示。



图 11.1-6　canvas 元素形成的绘图区域

11.2　Web 环境下基本图形的生成

由前面的介绍可以知道，要想在 Web 中实现基本的绘图功能，仅仅依靠 canvas 元素的绘图方法是不可能的。要想实现这一功能，必须得到计算机图形学相关算法的支持。可以

通过使用 JavaScript 脚本语言实现计算机图形学的相关算法,构造相关图形,实现交互绘图功能,实现在 Web 环境下生成基本图形。

11.2.1 直线的绘制

绘制直线是绘图操作中最常用的操作之一,通过对路径起点与终点的坐标来直接定义唯一直线的方法进行进一步的改进,可以实现通过鼠标直接控制直线的起点与终点,完成直线的绘制。相关功能代码如下所示:

```
var points = new Array();                                  //绘图的点集
function line(){
    huabu.onmousedown = function (e) {                    //在 canvas 中按下鼠标按钮时触发函数
        e = window.event||e;
        context.lineWidth = "2";
        context.strokeStyle = "red";
        var X = e.pageX - this.offsetLeft;                 //绘制直线的 x、y 范围
        var Y = e.pageY - this.offsetTop;
        var pois = Object.create({x:X,y:Y});               //构建绘制直线的点
        points.push(pois);                                 //将绘制直线的点加到绘图点集中
        if(points.length == 1) {
            return null;
        }else{//开始画直线
            context.beginPath();
            context.moveTo(points[0].x,points[0].y);       //直线起始点
            for(var i = 0;i < points.length;i++){
                context.lineTo(points[i].x,points[i].y); //直线的下一点
            }
            context.stroke();                              //绘制出点集的路径
        }
    }
}
```

添加名为“画线”的可以执行画直线功能的按钮:

```
<input type = "button" name = "huaxian" value = "画线" onclick = "line()" />
```

在定义的画布中,单击“画线”按钮,在任意位置单击两点,执行里面画直线功能的 JavaScript 脚本,形成线宽为 2 像素的红色直线。通过 JavaScript 绘制的直线效果如图 11.2-1 所示。

图 11.2-1 使用 JavaScript 绘制直线

11.2.2 封闭多边形的绘制

封闭多边形的绘制是在画直线功能的前提下执行的，在此基础上添加一个封闭按钮，将直线的最后一点与直线起始点连接，形成封闭多边形。执行封闭功能的代码如下：

```
function closeButton() {                  //封闭按钮
    var clo = points[0];                  //直线起始点
    points.push(clo);                     //执行封闭功能,将最后一点与起始点连接,形成封闭图形
    for(var cc = 0;cc < points.length;cc++) {      //重新执行画线功能,更新直线点集
        if(cc == 0) {
            context.beginPath();
            context.moveTo(points[0].x,points[0].y);
        }
    else{
            context.lineTo(points[cc].x,points[cc].y);
        }
    }
    context.stroke();
    points.length = 0;                    //清空整个画布中的点
}
```

添加名为“封闭按钮”的可以形成封闭多边形的按钮：

```
<input type = "button" name = "fengbi" value = "封闭按钮" onclick = "closeButton()" />
```

通过封闭按钮封闭图形是在画线功能的条件下形成的。单击“画线”按钮，在画布中随意绘制直线；此时若单击“封闭按钮”，便会执行封闭命令，将上面绘制直线的最后一点与绘制直线的起始点连接到一起，形成封闭多边形，如图 11.2-2 所示。

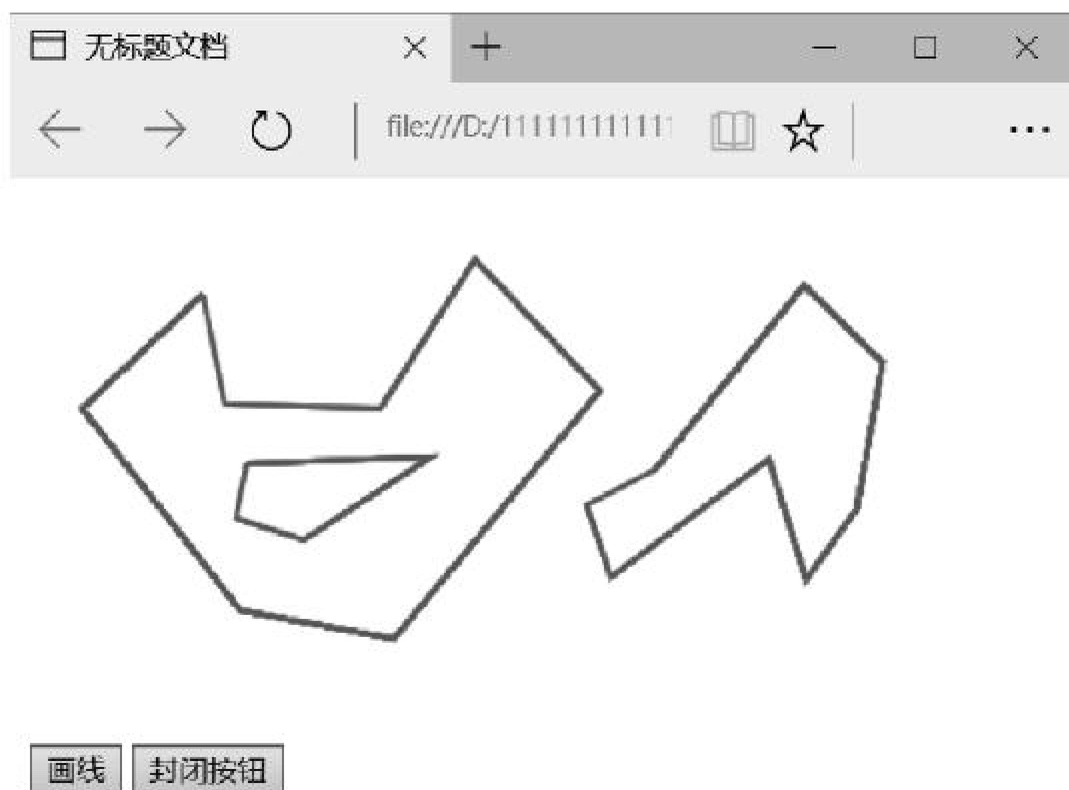


图 11.2-2 封闭多边形

需要注意的是，上述代码中 points.length＝0 的作用是清空整个画布中绘图的点集。在此之上的代码表示一个封闭多边形已经完成，若在此时不清空点集，再单击下一点，会以已经完成的封闭多边形的起始点为起点，继续执行绘制直线的功能，形成两个连接在一起的封闭图形。

11.2.3 多边形的扫描填充

基于 Web 环境形成的多边形扫描填充是计算机图形学基于 JavaScript 的实现。其编程思想来自前文中多边形的扫描转换的基本原理,即以 2 像素为宽度单位,在 y 方向由上而下地进行扫描,并根据交点的情况进行填充。当多边形共享顶点的两边分别落在扫描线两侧时,交点数量取 1；当多边形共享顶点的两边位于扫面线一侧时,交点数取 0 或 2,具体根据两条边的位置来定；如果同时落在扫面线上方,交点数取 2；若同时落在扫描线下方,交点数取 0。总体上由扫描线上共享一个顶点的两条边的另外两个端点的 y 值是否大于此共享顶点的 y 值来确定交点的取值。执行多边形填充的代码及部分代码功能解释如下：

```
function full(){
    var ymax = points[0].y;
    var ymin = points[0].y;
    var IpointsX = new Array();                         //扫描线与多边形的交点集

    for(var x = 0;x < points.length;++x){         //共享顶点两条边的另外两个端点的最大/小值
        if(points[x].y > ymax){
            ymax = points[x].y;
        }
    }
    for(var x = 0;x < points.length;++x){
        if(points[x].y < ymin){
            ymin = points[x].y;
        }
    }
    for(var i = ymin;i <= ymax;i++) {
        for(var j = 0;j < points.length - 1;j++) {
        //共享顶点的两条边分布在扫描线的两侧,取一个交点
            if((points[j].y < i)&&(points[j + 1].y > i)) {     //多边形与扫描线的交点
                var IX =
(points[j].x - points[j + 1].x) * (i - points[j + 1].y)/(points[j].y - points[j + 1].y) + points
[j + 1].x;
                IpointsX.push(IX);                      //将上述情况的交点放到交点集中
            }
            else if((points[j].y > i)&&(points[j + 1].y < i)){//k < 0,一个交点
                var IX =
(points[j].x - points[j + 1].x) * (i - points[j + 1].y)/(points[j].y - points[j + 1].y) + points
[j + 1].x;
                IpointsX.push(IX);
            }
            else if(i == points[j].y){                  //共享顶点在扫描线同一边
                if((j == 0)||(j == points.length - 1)){     //多边形外圈端点
                    //顶点的两边在扫描线下方,交点为 0
                    if((points[1].y > i)&&(points[points.length - 2].y < i)){
                        IpointsX.push(points[0].x);
                    }
                    //顶点的两边在扫描线上方,交点为 2
```

```
                    if((points[1].y<i)&&(points[points.length-2].y>i)){
                        IpointsX.push(points[0].x);
                    }
                }else{     //多边形内部端点
                    if((points[j-1].y>i)&&(points[j+1].y<i)){     //内部交点取0
                        IpointsX.push(points[j].x);
                    }
                    if((points[j-1].y<i)&&(points[j+1].y>i)){     //内部交点取2
                        IpointsX.push(points[j].x);
                    }
                }
            }
        }
        }                                                        //求交点结束
        IpointsX.sort(function(a,b){return a-b;});               //对交点进行排序
        for(var k=0;k<IpointsX.length;k=k+2){                    //使用2像素的直线进行连接,即填充
            context.lineWidth=2;
            context.beginPath();
            context.moveTo(IpointsX[k],i);
            context.lineTo(IpointsX[k+1],i);
            context.stroke();
        }
        IpointsX.length=0;
    }
}
```

添加名为“填充”的可以执行扫描填充功能的代码：

```
<input type="button" name="tianchong" value="填充" onclick="full()" />
```

在求出统一扫描线上的所有交点后，需要按照交点坐标的 x 值从小到大进行排序，并以两个一组进行配对，将配对后的点使用2像素的直线进行连接，即完成填充操作。最终的多边形扫描填充效果如图11.2-3所示。

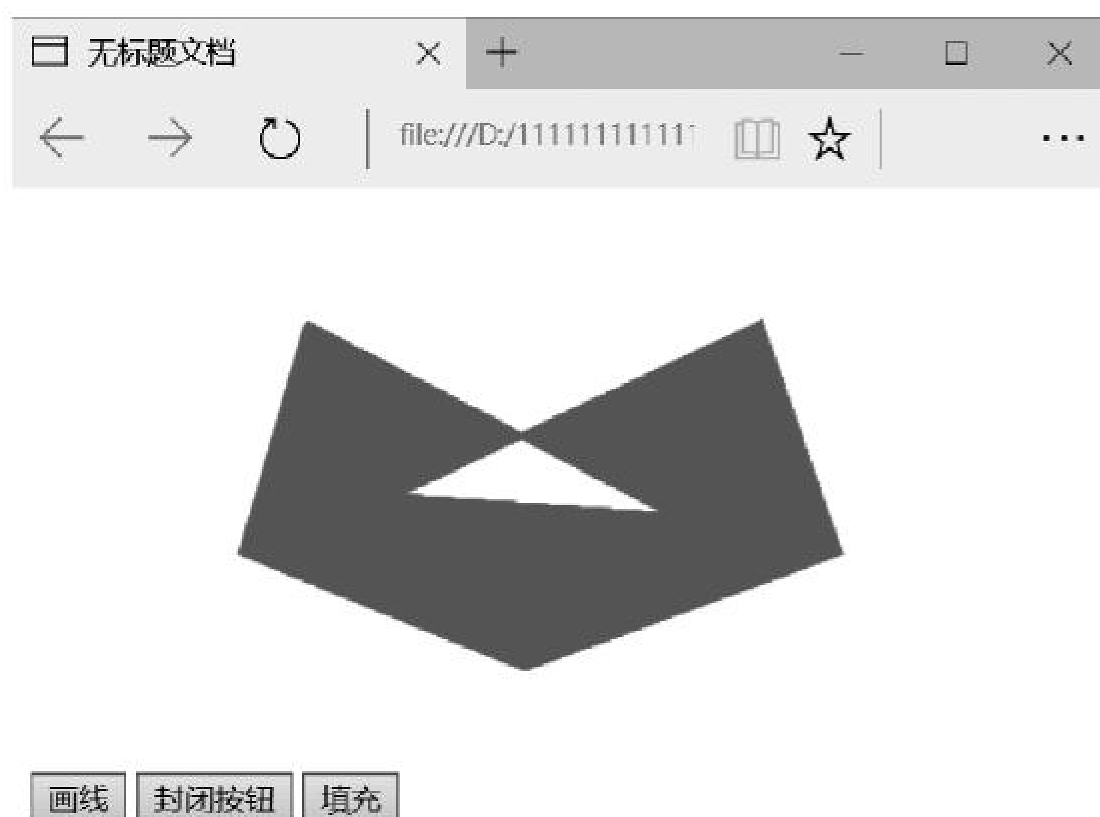


图 11.2-3 多边形扫描填充

需要注意的是，在进行填充时应将执行封闭多边形功能代码中的语句“points.length=0”去掉。因为此代码的意思是清空点集，如果保留该语句，那么在执行填充时，比较得到的

最大纵坐标 Y_{max} 及最小纵坐标 Y_{min} 也将被清空，不能完成填充操作。不仅填充时需要删除此代码语句，接下来的二维变换、三维变换也需删除该语句。

11.2.4 多边形的裁剪

多边形的裁剪就是用裁剪框对多边形进行裁剪形成新的图形。这里采用矩形裁剪框对多边形进行逐边裁剪，详细算法原理根据前面第 4 章的逐边裁剪法，以 S 和 P 分别表示多边形的某一边的起点和终点。执行多边形裁剪功能的相关代码如下：

```
function cut(){
        var sx;var sy;var fx;var fy;
        var Cutpoints = new Array();                      //裁剪点集
        var Rnum = 0;
        huabu.onmousedown = function(e){
            if(Rnum == 1){                                //单击两点确定裁剪框
            e = window.event||e;
            var x1 = e.pageX - this.offsetLeft;
            var y1 = e.pageY - this.offsetTop;
            sx = x1;sy = y1;
            }
        else if(Rnum == 2){
            e = window.event||e;
            var x2 = e.pageX - this.offsetLeft;
            var y2 = e.pageY - this.offsetTop;
            fx = x2;fy = y2;
            var Xmin,Ymin,Xmax,Ymax;                      //裁剪框点中的最大值与最小值
            if(sx > fx) {Xmin = fx; Xmax = sx; }
            else{Xmin = sx; Xmax = fx; }
            if(sy > fy) {Ymin = fy; Ymax = sy; }
            else{Ymin = sy; Ymax = fy; }
                points.push(points[0]);                   //两点首尾相接，得到裁剪框
                for(var l = 0;l < points.length - 1;l++){//以 Xmin 为裁剪边的切割情况
                    if((points[l].x > Xmin)&&(points[l + 1].x > Xmin))
                    {//S、P 均在裁剪框可见侧，多边形完全可见，点全部保存
                        Cutpoints.push(points[l + 1]);
                    }
                    else if((points[l].x < Xmin)&&(points[l + 1].x > Xmin))
                    {//S 在裁剪边外侧，P 在内侧，进入可见侧，多边形部分可见
                        var a = points[l + 1].y - points[l].y;
                        var b = points[l].x - points[l + 1].x;
                        var c = points[l + 1].x * points[l].y - points[l].x * points[l + 1].y;
                        var CutY = Math.round( - (a * Xmin + c)/b);
                        var CutX = Xmin;
                        var cp = Object.create({x:CutX,y:CutY});      //S、P 与裁剪边的交点
                        Cutpoints.push(cp);
                        Cutpoints.push(points[l + 1]);
                    }
                    else if((points[l].x > Xmin)&&(points[l + 1].x < Xmin))
                    {//S 在裁剪边内侧，P 在外侧，进入不可见侧，多边形部分可见
```

```
            var a = points[l + 1].y - points[l].y;
            var b = points[l].x - points[l + 1].x;
            var c = points[l + 1].x * points[l].y - points[l].x * points[l + 1].y;
            var CutY = Math.round( - (a * Xmin + c)/b);
            var CutX = Xmin;
            var cp = Object.create({x:CutX,y:CutY});     //交点
            Cutpoints.push(cp);
        }
}
points.length = 0;  //将 points 点集清空,全部放入以 Xmin 为裁剪边的交点集
                    //Cutpoints 的点,再清空交点集 Cutpoints,对矩形裁剪框
                    //的其他裁剪边进行讨论
for(var j = 0;j < Cutpoints.length;j++){
    points.push(Cutpoints[j]);
}
Cutpoints.length = 0;
points.push(points[0]);                    //首尾相接,封闭
for(var l1 = 0;l1 < points.length - 1;l1++){  //以 Xmax 为裁剪边的切割情况
    if((points[l1].x < Xmax)&&(points[l1 + 1].x < Xmax))
    {    //S、P 均在裁剪边的内侧,多边形完全可见,点全部保存
        Cutpoints.push(points[l1 + 1]);
    }
    else if((points[l1].x > Xmax)&&(points[l1 + 1].x < Xmax))
    {//S 在裁剪边外侧,P 在内侧,进入可见侧,多边形部分可见
        var a = points[l1 + 1].y - points[l1].y;
        var b = points[l1].x - points[l1 + 1].x;
        var c = points[l1 + 1].x * points[l1].y - points[l1].x * points[l1 + 1].y;
        var CutY = Math.round( - (a * Xmax + c)/b);
        var CutX = Xmax;
        var cp = Object.create({x:CutX,y:CutY});  //交点
        Cutpoints.push(cp);
        Cutpoints.push(points[l1 + 1]);
    }
    else if((points[l1].x < Xmax)&&(points[l1 + 1].x > Xmax))
    {//S 在裁剪边内侧,P 在外侧,进入不可见侧,多边形部分可见
        var a = points[l1 + 1].y - points[l1].y;
        var b = points[l1].x - points[l1 + 1].x;
        var c = points[l1 + 1].x * points[l1].y - points[l1].x * points[l1 + 1].y;
        var CutY = Math.round( - (a * Xmax + c)/b);
        var CutX = Xmax;
        var cp = Object.create({x:CutX,y:CutY});  //交点
        Cutpoints.push(cp);
    }
}
points.length = 0;
for(var j1 = 0;j1 < Cutpoints.length;j1++){
    points.push(Cutpoints[j1]);
}
Cutpoints.length = 0;
points.push(points[0]);
for(var l2 = 0;l2 < points.length - 1;l2++){     //以 Ymin 为裁剪边的切割情况
```

```
        if((points[l2].y > Ymin)&&(points[l2 + 1].y > Ymin))
        {//S、P 均在裁剪边内侧,多边形完全可见
            Cutpoints.push(points[l2 + 1]);
        }
        else if((points[l2].y < Ymin)&&(points[l2 + 1].y > Ymin))
        {//S 在裁剪边外侧,P 在内侧,进入可见侧,多边形部分可见
            var a = points[l2 + 1].y - points[l2].y;
            var b = points[l2].x - points[l2 + 1].x;
            var c = points[l2 + 1].x * points[l2].y - points[l2].x * points[l2 + 1].y;
            var CutX = Math.round( - (b * Ymin + c)/a);
            var CutY = Ymin;
            var cp = Object.create({x:CutX,y:CutY});   //交点
            Cutpoints.push(cp);
            Cutpoints.push(points[l2 + 1]);
        }
        else if((points[l2].y > Ymin)&&(points[l2 + 1].y < Ymin))
        {//S 在裁剪边内侧,P 在外侧,进入不可见侧,多边形部分可见
            var a = points[l2 + 1].y - points[l2].y;
            var b = points[l2].x - points[l2 + 1].x;
            var c = points[l2 + 1].x * points[l2].y - points[l2].x * points[l2 + 1].y;
            var CutX = Math.round( - (b * Ymin + c)/a);
            var CutY = Ymin;
            var cp = Object.create({x:CutX,y:CutY});   //交点
            Cutpoints.push(cp);
        }
    }
    points.length = 0;
    for(var j2 = 0;j2 < Cutpoints.length;j2++) {
        points.push(Cutpoints[j2]);
    }
    Cutpoints.length = 0;
    points.push(points[0]);
    for(var l3 = 0;l3 < points.length - 1;l3++){     //以 Ymax 为裁剪边的切割情况
        if((points[l3].y < Ymax)&&(points[l3 + 1].y < Ymax))
        {//S、P 均在裁剪边内侧,多边形完全可见
            Cutpoints.push(points[l3 + 1]);
        }
        else if((points[l3].y > Ymax)&&(points[l3 + 1].y < Ymax))
        {//S 在裁剪边外侧,P 在内侧,进入可见侧,多边形部分可见
            var a = points[l3 + 1].y - points[l3].y;
            var b = points[l3].x - points[l3 + 1].x;
            var c = points[l3 + 1].x * points[l3].y - points[l3].x * points[l3 + 1].y;
            var CutX = Math.round( - (b * Ymax + c)/a);
            var CutY = Ymax;
            var cp = Object.create({x:CutX,y:CutY});   //交点
            Cutpoints.push(cp);
            Cutpoints.push(points[l3 + 1]);
        }
        else if((points[l3].y < Ymax)&&(points[l3 + 1].y > Ymax))
        {//S 在裁剪边外侧,P 在内侧,进入不可见侧,多边形部分可见
            var a = points[l3 + 1].y - points[l3].y;
```

```
                var b = points[l3].x - points[l3 + 1].x;
                var c = points[l3 + 1].x * points[l3].y - points[l3].x * points[l3 + 1].y;
                var CutX = Math.round( - (b * Ymax + c)/a);
                var CutY = Ymax;
                var cp = Object.create({x:CutX,y:CutY});   //交点
                Cutpoints.push(cp);
            }
        }
        points.length = 0;
        for(var j3 = 0;j3 < Cutpoints.length;j3++){
            points.push(Cutpoints[j3]);
        }//结束
        //清空画布中的多边形的点,更新成裁剪后形成的新的多边形
        context.clearRect(0,0,mycanvas.width,mycanvas.height);
        points.push(points[0]);
        context.beginPath();
        for(var cc = 0;cc < points.length;cc++){
            if(cc == 0) {
                context.beginPath();
                context.moveTo(points[0].x,points[0].y);
            }else{
                context.lineTo(points[cc].x,points[cc].y);
            }
        }
        context.strokeStyle = "black";
        context.stroke();
    }
  }
}
```

添加名为“裁剪”的可以执行裁剪功能的按钮:

```
<input type = "button" name = "caijian" value = "裁剪" onclick = "cut()" />
```

单击“裁剪”按钮,在任意地方单击两点,形成矩形裁剪框,便执行裁剪功能形成新的多边形。运行效果和裁剪后多边形的扫描变换如图 11.2-4 所示。

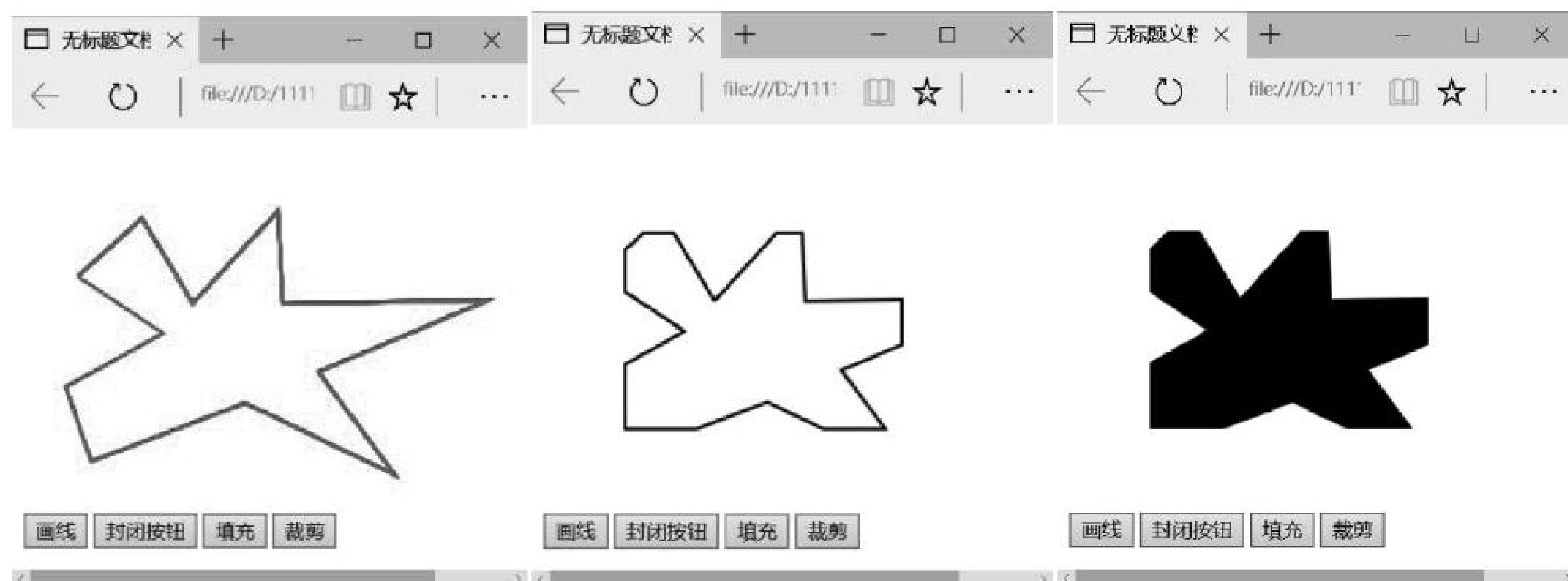


图 11.2-4 矩形裁剪框裁剪多边形

11.2.5 二维图形的组合变换

二维图形的基本变换主要包括比例旋转、镜像、错移、缩放、平移等操作。在获知图形关键点坐标的情况下，首先将二维图形的点坐标变换为齐次坐标，即用三维向量表示二维空间点。利用齐次坐标将二维变换矩阵形成统一的形式，统一形式为

$$[x^* \quad y^* \quad 1] = [x \quad y \quad 1]\begin{bmatrix} a & b & p \\ c & d & q \\ l & m & s \end{bmatrix}$$

变换矩阵为

$$\boldsymbol{T} = \begin{bmatrix} a & b & p \\ c & d & q \\ l & m & s \end{bmatrix}$$

变换矩阵 $\boldsymbol{T}$ 中左上角的 2×2 子矩阵包含 a、b、c、d 四个元素，可使图形产生比例、镜像、旋转等基本变换。比例变换由 a、d 元素控制，二者分别决定了图形上任意一点的 x 坐标与 y 坐标的放大(或缩小)倍数，在作等比例变换时，需要将二者与相同大小的系数相乘。镜像变换由 a、d 的正负决定，当 a 的正负改变时，图形以 x 轴为镜像线作镜像变换；当 d 的正负改变时，图形以 y 轴为镜像线作镜像变换；当二者同时改变时，图形沿原点作镜像变换。旋转变换由 a、b、c、d 四个元素共同确定。左下角的 1×2 子矩阵，包括元素 l、m，可使图形产生平移变换，l、m 分别是 x 向和 y 向的平移量，在作其他变化时需使 $l=0$，$m=0$。

1. 二维图形的平移变换

多边形可以看成是点的集合，所以图形变换可以看作是图形上的点进行坐标变换。平移变换以原点为基准，即将原图形的点分别向 x 向、y 向进行平移变换，得到新的齐次坐标。执行二维图形平移变换功能的代码如下所示：

```
function move(){
    var arr = new Array;                        //平移后的点集
    var num = 0;
    var sX = 0;var fX = 0;var sY = 0;var fY = 0;
    huabu.onmousedown = function(e){
        e = window.event||e;
        num++;
        if(num == 1){//平移前的点
            fX = e.pageX - this.offsetLeft;
            fY = e.pageY - this.offsetTop;
        }
        else if(num == 2){//平移后的点
            sX = e.pageX - this.offsetLeft;
            sY = e.pageY - this.offsetTop;
            var dX = sX - fX;                   //图形平移的距离
            var dY = sY - fY;
            for(var a = 0;a < points.length;a++){    //平移前多边形的点
```

```
                //平移后的点的集合,将原多边形以点的形式进行整体平移
                var newPoints = Object.create({x:dX + points[a].x,y:dY + points[a].y});
                arr.push(newPoints);
            }
            for(var b = 0;b < points.length;b++){          //平移后多边形的点
                points[b] = arr[b];                        //替换原图形
            }
            context.clearRect(0,0,mycanvas.width,mycanvas.height);
            for(var cc = 0;cc < points.length;cc++) {      //更新点集
                if(cc == 0) {
                    context.beginPath();
                    context.moveTo(points[0].x,points[0].y);
                }else{
                    context.lineTo(points[cc].x,points[cc].y);
                }
            }
            context.strokeStyle = "black";
            context.stroke();
        }
    }
}
```

添加名为“平移”的可以执行二维图形平移变换功能的按钮,效果如图 11.2-5 所示。

```
< input type = "button" name = "pingyi" value = "平移" onclick = "move()" />
```

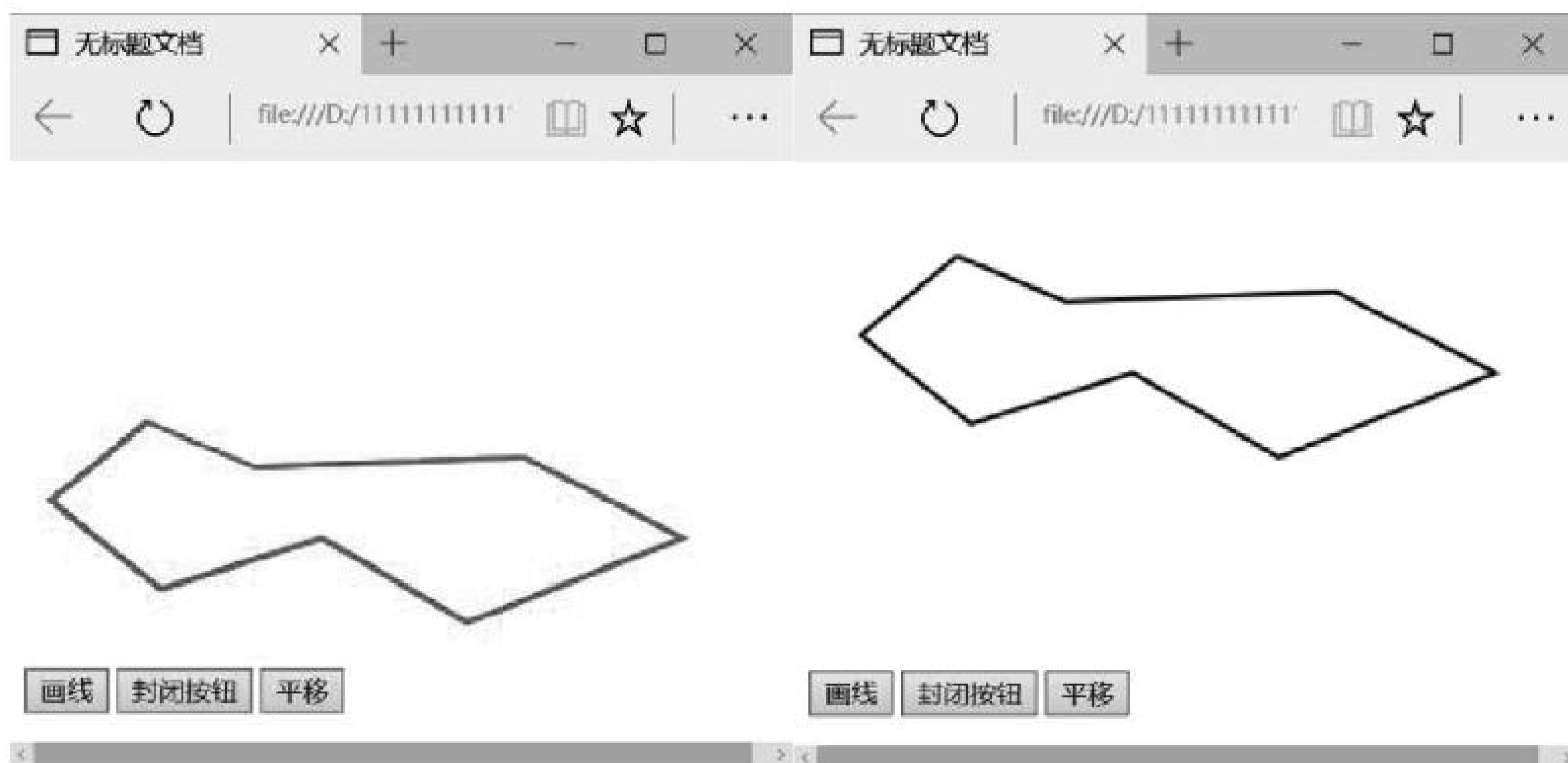


图 11.2-5 二维图形的平移变换

2. 二维图形的旋转变换

二维图形的旋转变换以原点为基准。由 5.2 节二维组合变换可知,二维图形旋转需要通过几个基本变换的级联,实现组合变换。对于图形的旋转,需要先将图形平移,使旋转点与原点重合,将原图形平移到基本变换位置,设此操作的变换矩阵为 $\boldsymbol{T}_1$;再将图形绕原点旋转 θ,得到新的中间图形,设此操作的变换矩阵为 $\boldsymbol{T}_2$;最后将旋转后的图形平移至原位置,设此操作的变换矩阵为 $\boldsymbol{T}_3$。此时得到的便是绕旋转点旋转 θ 后得到的图形。

将以上三个变换矩阵相乘，即可得到绕任意点旋转的组合变换矩阵 $\boldsymbol{T}$：

$$\boldsymbol{T}=\boldsymbol{T}_1\boldsymbol{T}_2\boldsymbol{T}_3=\begin{bmatrix}1&0&0\\0&1&0\\-x_0&-y_0&1\end{bmatrix}\begin{bmatrix}\cos\theta&\sin\theta&0\\-\sin\theta&\cos\theta&0\\0&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\x_0&y_0&1\end{bmatrix}$$

可以执行旋转功能的相关代码如下所示：

```
function RoelveButton(){//旋转按钮,调用旋转函数
            //调取 id 为 re 的功能块,相关数据功能均在此功能块中
            document.getElementById("re").style.display = "";
            huabu.onmousedown = function(e){
                e = window.event||e;
                var rx1 = e.pageX - this.offsetLeft;     //随意选取一点
                var ry1 = e.pageY - this.offsetTop;
                //将选取点的值赋给 Xzuobiao、Yzuobiao,作为旋转点
                document.getElementById("Xzuobiao").value = rx1;
                document.getElementById("Yzuobiao").value = ry1;
            }
        }
function revolve(){//旋转函数
            var ang = document.getElementById("drevolve").value;     //旋转角度
            var zheng = Math.sin(ang * Math.PI/180);     //正弦
            var yu = Math.cos(ang * Math.PI/180);        //余弦
            huabu.onmousedown = function(e){
                e = window.event||e;
                var rx1 = e.pageX - this.offsetLeft;     //拾取点(rx1,ry1)
                var ry1 = e.pageY - this.offsetTop;
                //将该点的值通过查找 id 的方式赋给旋转点(rx,ry)
                document.getElementById("Xzuobiao").value = rx1;
                document.getElementById("Yzuobiao").value = ry1;
                }
                var rx = document.getElementById("Xzuobiao").value;
                var ry = document.getElementById("Yzuobiao").value;
            var T1 = new Array(3);                    //将图形旋转点(rx,ry)平移到原点位置
            for(var i = 0;i < T1.length;i++){
                T1[i] = new Array(3);                 //平移变换矩阵用二维数组表示
            }
            T1[0][0] = 1; T1[0][1] = 0; T1[0][2] = 0;
            T1[1][0] = 0; T1[1][1] = 1; T1[1][2] = 0;
            T1[2][0] = - rx; T1[2][1] = - ry; T1[2][2] = 1;
            var T2 = new Array(3);                    //将图形绕旋转点旋转
            for(var i = 0;i < T2.length;i++){
                T2[i] = new Array(3);
            }
            T2[0][0] = yu; T2[0][1] = zheng; T2[0][2] = 0;
            T2[1][0] = - zheng; T2[1][1] = yu; T2[1][2] = 0;
            T2[2][0] = 0;T2[2][1] = 0; T2[2][2] = 1;
            var T3 = new Array(3);                   //将旋转后的图形再平移到原来的位置上
            for(var i = 0;i < T3.length;i++){
                T3[i] = new Array(3);
            }
```

```
            T3[0][0] = 1; T3[0][1] = 0; [0][2] = 0;
            T3[1][0] = 0; T3[1][1] = 1; T3[1][2] = 0;
            T3[2][0] = rx; T3[2][1] = ry; T3[2][2] = 1;
            var temp1 = Multiply2D3v3(T1,T2);                    //定义 3×3 矩阵
            var T = Multiply2D3v3(temp1,T3);                     //总体变换矩阵
            var Revpoints = new Array();                         //旋转后的点的集合
            for(var i = 0;i < points.length;i++){
                var sarrpoints = new Array(1);
                sarrpoints[0] = [points[i].x,points[i].y,1];     //[x   y   1]矩阵
                var farrpoints = Multiply2D1v3(sarrpoints,T);    //定义 1×3 矩阵
                var newP = Object.create({x:farrpoints[0][0],y:farrpoints[0][1]});
                Revpoints.push(newP);
            }
            points.length = 0;
            for(var ii = 0;ii < Revpoints.length;ii++) {
                points.push(Revpoints[ii]);                      //更新点集
            }
            context.clearRect(0,0,mycanvas.width,mycanvas.height);
            context.beginPath();
            context.moveTo(points[0].x,points[0].y);
            for(var l = 1;l < points.length;l++){
                context.lineTo(points[l].x,points[l].y);
            }
            context.strokeStyle = "black";
            context.stroke();
    }
//定义 3×3 矩阵
function Multiply2D3v3(arr1,arr2){
        var tempArray = new Array(3);
        for(var i = 0;i < 3;i++){
            tempArray[i] = new Array(3);
        }
        if(arr1[0].length!= arr2.length)     //若相乘矩阵不都是 3 阶时,提出警告
        {
            alert("数组行列不匹配,无法相乘");
        }else{
tempArray[0][0] = arr1[0][0] * arr2[0][0] + arr1[0][1] * arr2[1][0] + arr1[0][2] * arr2[2][0];
tempArray[0][1] = arr1[0][0] * arr2[0][1] + arr1[0][1] * arr2[1][1] + arr1[0][2] * arr2[2][1];
tempArray[0][2] = arr1[0][0] * arr2[0][2] + arr1[0][1] * arr2[1][2] + arr1[0][2] * arr2[2][2];
tempArray[1][0] = arr1[1][0] * arr2[0][0] + arr1[1][1] * arr2[1][0] + arr1[1][2] * arr2[2][0];
tempArray[1][1] = arr1[1][0] * arr2[0][1] + arr1[1][1] * arr2[1][1] + arr1[1][2] * arr2[2][1];
tempArray[1][2] = arr1[1][0] * arr2[0][2] + arr1[1][1] * arr2[1][2] + arr1[1][2] * arr2[2][2];
tempArray[2][0] = arr1[2][0] * arr2[0][0] + arr1[2][1] * arr2[1][0] + arr1[2][2] * arr2[2][0];
tempArray[2][1] = arr1[2][0] * arr2[0][1] + arr1[2][1] * arr2[1][1] + arr1[2][2] * arr2[2][1];
tempArray[2][2] = arr1[2][0] * arr2[0][2] + arr1[2][1] * arr2[1][2] + arr1[2][2] * arr2[2][2];
        return tempArray;
        }
    }
    //定义 1×3 矩阵
    function Multiply2D1v3(arr1,arr2){
        var tempArray = new Array(1);
```

```
            tempArray[0] = new Array(3);
        if(arr1[0].length!= arr2.length)
        {
            alert("数组行列不匹配,无法相乘");
        }else{
tempArray[0][0] = arr1[0][0] * arr2[0][0] + arr1[0][1] * arr2[1][0] + arr1[0][2] * arr2[2][0];
tempArray[0][1] = arr1[0][0] * arr2[0][1] + arr1[0][1] * arr2[1][1] + arr1[0][2] * arr2[2][1];
tempArray[0][2] = arr1[0][0] * arr2[0][2] + arr1[0][1] * arr2[1][2] + arr1[0][2] * arr2[2][2];
        return tempArray;
        }
}
```

添加名为“旋转”的可以执行旋转功能的按钮，并添加块级元素 div，使实现旋转功能的相关数据均在其中表示。

```
<input type = "button" name = "xuanzhuan" value = "旋转" onclick = "RoelveButton()" />
<div id = "re">
旋转角度<input type = "text" value = "45" id = "drevolve" size = "1"/>
旋转中心 X<input type = "text" id = "Xzuobiao" size = "1"/>
Y<input type = "text" id = "Yzuobiao" size = "1"/>
<input type = "button" value = "确定" onclick = "revolve()" />
</div>
```

当完成多边形后，单击“旋转”按钮，调用 revolve()函数，调用旋转功能，在页面中随意单击一点作为旋转点，此时在旋转中心 x、y 中便会出现旋转点的值。再单击“确定”按钮，调用 RoelveButton()函数，完成旋转。旋转角度经设置默认为 45°，也可以根据需要进行更改。二维图形的旋转变换效果如图 11.2-6 所示。

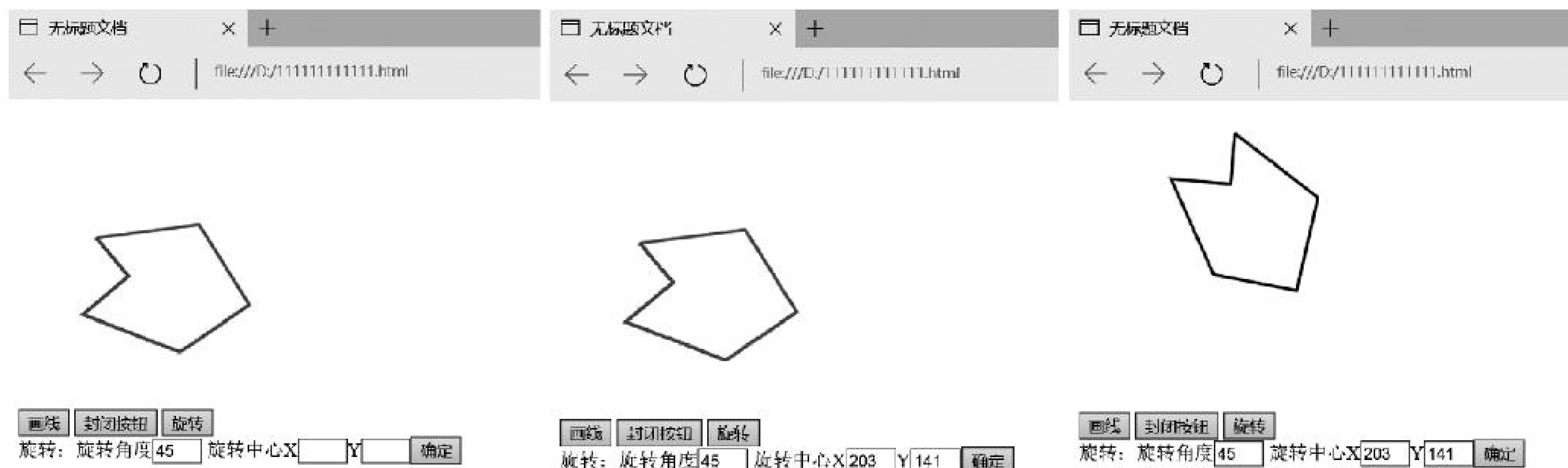


图 11.2-6　二维图形的旋转变换

3. 二维图形的比例变换

二维图形的比例变换是以原点为基准。对于图形的比例变换，需先将图形平移，使缩放中心与原点重合，设此操作的变换矩阵为 $\boldsymbol{T}_1$；在原点进行比例变换操作，设此操作的变换矩阵为 $\boldsymbol{T}_2$；再将变换后的图形平移至原位置，设此操作的变换矩阵为 $\boldsymbol{T}_3$。此时的图形便是在缩放中心缩放后得到的图形。相关功能代码如下所示：

```
function zoomButton(){
        document.getElementById("ddzoom").style.display = "";
        huabu.onmousedown = function(e){
```

```
            e = window.event||e;
            var Zoomx1 = e.pageX - this.offsetLeft;
            var Zoomy1 = e.pageY - this.offsetTop;
            document.getElementById("ZoomX").value = Zoomx1;
            document.getElementById("ZoomY").value = Zoomy1;
            }
        }
function zoom( ){                                              //缩放函数
        huabu.onmousedown = function(e){
            e = window.event||e;
            var Zoomx1 = e.pageX - this.offsetLeft;            //缩放点
            var Zoomy1 = e.pageY - this.offsetTop;
            //将此点的值赋给 id 为 ZoomX、ZoomY 的缩放中心
            document.getElementById("ZoomX").value = Zoomx1;
            document.getElementById("ZoomY").value = Zoomy1;
            }
            var R = document.getElementById("ZoomR").value;  //比例因子
            var eX = document.getElementById("ZoomX").value; //缩放中心
            var eY = document.getElementById("ZoomY").value;
            var zoompoints = new Array();
            var T1 = new Array(3);
            for(var i = 0;i < T1.length;i++){                  //将图形的缩放中心平移至原点
                T1[i] = new Array(3);                          //平移变换矩阵用二维数组表示
            }
            T1[0][0] = 1; T1[0][1] = 0; T1[0][2] = 0;
            T1[1][0] = 0; T1[1][1] = 1; T1[1][2] = 0;
            T1[2][0] = - eX; T1[2][1] = - eY; T1[2][2] = 1;
            for(var i = 0;i < points.length;i++){              //进行平移操作
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points[i].x,points[i].y,1];       //[x  y  1]矩阵
            var farrpoints = Multiply2D1v3(sarrpoints,r1);     //定义 1×3 矩阵
            var newP = Object.create({x:farrpoints[0][0],y:farrpoints[0][1]});
            zoompoints.push(newP);                             //平移后点的集合
            }
            for(var j = 0;j < zoompoints.length;j++){          //进行缩放操作
                zoompoints[j].x = zoompoints[j].x * R;
                zoompoints[j].y = zoompoints[j].y * R;
            }
            var T3 = new Array(3);                         //将缩放后的图形平移至原来的缩放点
            for(var iii = 0;iii < r3.length;iii++){
                T3[iii] = new Array(3);
            }
            T3[0][0] = 1; T3[0][1] = 0; T3[0][2] = 0;
            T3[1][0] = 0; T3[1][1] = 1; T3[1][2] = 0;
            T3[2][0] = eX; T3[2][1] = eY; T3[2][2] = 1;
            var zoompoints1 = new Array();                     //进行平移操作
            for(var k = 0;k < points.length;k++) {
                var sarrpoints1 = new Array(1);
                sarrpoints1[0] = [zoompoints[k].x,zoompoints[k].y,1];
                var farrpoints1 = Multiply2D1v3(sarrpoints1,r3);
                var newP = Object.create({x:farrpoints1[0][0],y:farrpoints1[0][1]});
```

```
            zoompoints1.push(newP);
        }
        points.length = 0;
        for(var c = 0;c < zoompoints1.length;c++)          //进行点的更新
        {
            points.push(zoompoints1[c]);
        }
        context.clearRect(0,0,mycanvas.width,mycanvas.height);
        context.beginPath();
        context.moveTo(points[0].x,points[0].y);
        for(var l = 1;l < points.length;l++) {
            context.lineTo(points[l].x,points[l].y);
        }
        context.stroke();
    }
```

添加名为“缩放”的可以执行缩放功能的按钮，并添加块级元素 div，使实现比例变换的相关数据均在其中表示。

```
<input type = "button" name = "suofang" value = "缩放" onclick = "zoomButton()" />
<div id = "ddzoom">
比例因子<input type = "text" value = "0.5" id = "ZoomR" size = "1" />
缩放中心 X<input type = "text" id = "ZoomX" size = "1" />
Y<input type = "text" id = "ZoomY" size = "1" />
<input type = "button" value = "确定" onclick = "zoom()" />
</div>
```

当完成多边形后，单击“缩放”按钮，调用 zoom()函数，调用比例缩放功能，随意单击一点作为缩放点，此时在缩放中心 x、y 中便会出现缩放点的值。再单击“确定”按钮，调用 zoomButton()函数，完成比例缩放。比例因子经设置默认为 0.5，也可以根据需要进行更改。二维图形的比例变换效果如图 11.2-7 所示。

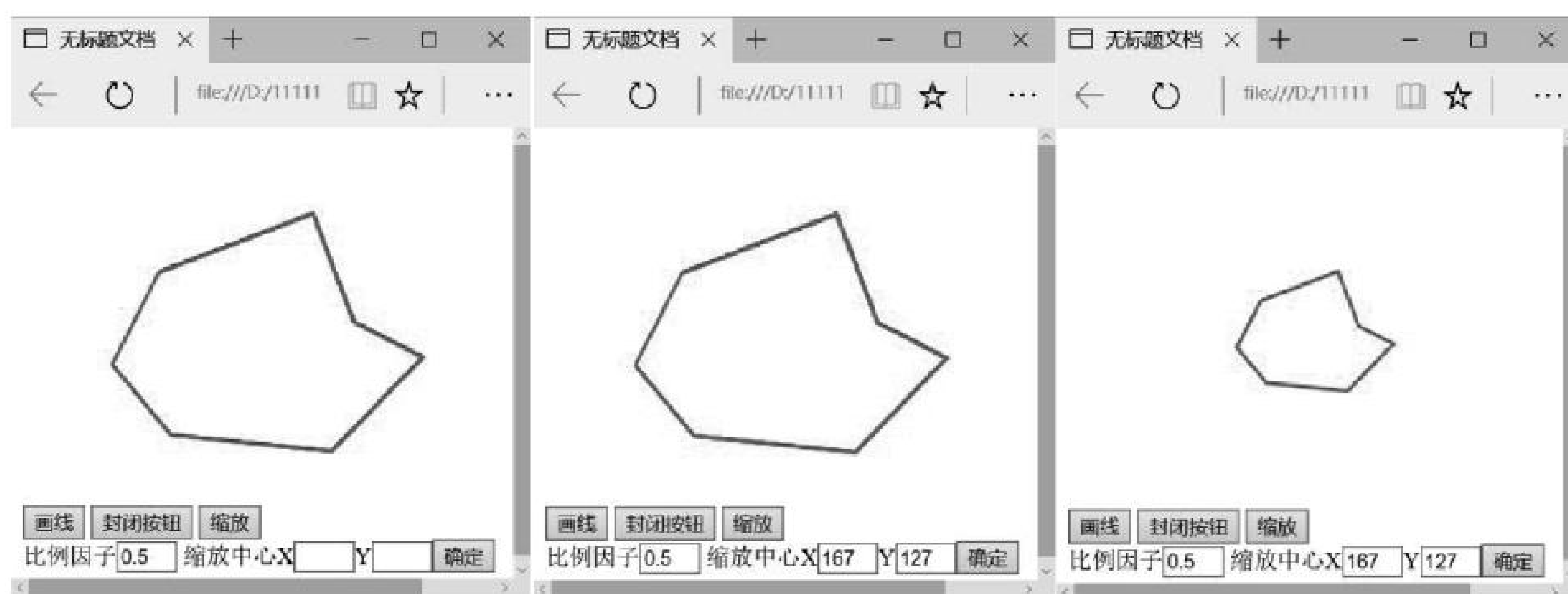


图 11.2-7 二维图形的比例变换

4. 二维图形的镜像变换

二维图形的镜像变换是以原点为基准。对于图形的镜像变换，需先将图形与镜像线进行平移，使镜像线和 y 轴的交点与原点重合，设此操作的变换矩阵为 $\boldsymbol{T}_1$；绕原点旋转 θ，使镜像线与 x 轴重合，设此操作的变换矩阵为 $\boldsymbol{T}_2$；在 x 轴进行镜像变换，设此操作的变换矩

阵为 T_3；绕原点反向旋转 θ,至原位置,设此操作的变换矩阵为 T_4；再平移到图形变换前的位置,设此操作的变换矩阵为 T_5。此时的图形便是镜像后的图形。相关功能代码如下所示：

```
function mirror(){
        alert('请点击两点确定镜像直线');
        var shu = 0;
        var mirLine = new Array(2);                    //镜像线的点的集合
        var Mirpoints = new Array();                   //镜像后的点的集合
        huabu.onmousedown = function(e){               //单击两点确定镜像直线
            e = window.event||e;
            sX = e.pageX - this.offsetLeft;
            sY = e.pageY - this.offsetTop;
            var mP = Object.create({x:sX,y:sY});
            if(shu == 1){
                mirLine[1] = mP;
            }else{
                mirLine[0] = mirLine[1];
                mirLine[1] = mP;
        //旋转角度θ
        var ang = Math.atan((mirLine[1].y - mirLine[0].y)/(mirLine[1].x - mirLine[0].x));
        var T1 = new Array(3);                         //将图形与镜像直线进行平移
        for(var i = 0;i < T1.length;i++){
            T1[i] = new Array(3);
        }
        T1[0][0] = 1; T1[0][1] = 0; T1[0][2] = 0;
        T1[1][0] = 0; T1[1][1] = 1; T1[1][2] = 0;
        T1[2][0] = 0; T1[2][1] =
    (mirLine[1].y - mirLine[0].y)/(mirLine[1].x - mirLine[0].x) * mirLine[0].x - mirLine[0].y;
        T1[2][2] = 1;
        var T2 = new Array(3);                   //绕坐标原点旋转θ,使镜像线与x轴重合
        for(var i = 0;i < T2.length;i++){
            T2[i] = new Array(3);
        }
        T2[0][0] = Math.cos( - ang);T2[0][1] = Math.sin( - ang); T2[0][2] = 0;
        T2[1][0] = - Math.sin( - ang); T2[1][1] = Math.cos( - ang); T2[1][2] = 0;
        T2[2][0] = 0; T2[2][1] = 0; T2[2][2] = 1;
        var T3 = new Array(3);                         //对x轴进行镜像变换
        for(var i = 0;i < T3.length;i++){
            T3[i] = new Array(3);
        }
        T3[0][0] = 1; T3[0][1] = 0; T3[0][2] = 0;
        T3[1][0] = 0; T3[1][1] = - 1; T3[1][2] = 0;
        T3[2][0] = 0; T3[2][1] = 0; T3[2][2] = 1;
        var T4 = new Array(3);                         //反向旋转至原位置
        for(var i = 0;i < T4.length;i++){
            T4[i] = new Array(3);
        }
        T4[0][0] = Math.cos(ang); T4[0][1] = Math.sin(ang); T4[0][2] = 0;
        T4[1][0] = - Math.sin(ang); T4[1][1] = Math.cos(ang); T4[1][2] = 0;
        T4[2][0] = 0; T4[2][1] = 0; T4[2][2] = 1;
        var T5 = new Array(3);                         //平移至原位置
        for(var i = 0;i < T5.length;i++) {
```

```
                T5[i] = new Array(3);
            }
            T5[0][0] = 1; T5[0][1] = 0; T5[0][2] = 0;
            T5[1][0] = 0; T5[1][1] = 1; T5[1][2] = 0;
            T5[2][0] = 0; T5[2][1] =
mirLine[0].y - (mirLine[1].y - mirLine[0].y)/(mirLine[1].x - mirLine[0].x) * mirLine[0].x;
            T5[2][2] = 1;
            var t12 = Multiply2D3v3(T1,T2);
            var t23 = Multiply2D3v3(t12,T3);
            var t34 = Multiply2D3v3(t23,T4);
            var t45 = Multiply2D3v3(t34,T5);                //得镜像的组合变换矩阵
            for(var i = 0;i < points.length;i++){           //进行点的组合变换
                var sarrpoints = new Array(1);
                sarrpoints[0] = [points[i].x,points[i].y,1];
                var farrpoints = Multiply2D1v3(sarrpoints,t45);
                var newP = Object.create({x:farrpoints[0][0],y:farrpoints[0][1]});
                Mirpoints.push(newP);
            }
            points.length = 0;
            for(var ii = 0;ii < Mirpoints.length;ii++){     //进行点的更新
                points.push(Mirpoints[ii]);
            }
            context.clearRect(0,0,mycanvas.width,mycanvas.height);
            context.beginPath();
            context.moveTo(points[0].x,points[0].y);
            for(var l = 1;l < points.length;l++){
                context.lineTo(points[l].x,points[l].y);
            }
            context.stroke();
            }
        }
    }
```

添加名为“镜像”的可以执行镜像功能的按钮。

```
<input type = "button" name = "jingxiang" value = "镜像" onclick = "mirror()" />
```

当完成多边形后，单击“镜像”按钮，便会弹出提示“请点击两点确定镜像直线”的对话框，单击“确定”按钮后单击两点确定镜像线，便完成镜像变换，形成镜像后的图形。二维图形的镜像变换效果如图 11.2-8 所示。



图 11.2-8　二维图形的镜像变换

11.2.6 二维图形拉伸生成三维图形

当给定或者拾取一个平面图形后，可通过拉伸得到三维立体图形。拉伸的意义在于直接给原始的二维坐标进行第三维度坐标的赋值，使其具有深度意义，实现图形的立体感。相对于二维图形，三维图形多出了深度维的坐标，因此，在换算成齐次坐标后，三维点即可以用4×1矩阵来表示。二维图形拉伸生成三维图形的相关代码如下所示：

```
var points3Dq = new Array();                           //三维变换前图形点集
var points3Dh = new Array();                           //三维变换后图形点集
function lashen(){
        var Zvalue = window.prompt("请输入拉伸距离",30);
        var tem1points = new Array();                  //拉伸前的点集
        var tem2points = new Array();                  //拉伸后的点集
        var lashen1 = new Array(4);
        for(var i = 0;i < lashen1.length;i++){         //将二维图形拉伸生成三维图形的矩阵变换
            lashen1[i] = new Array(4);
        }
        lashen1[0][0] = 1;lashen1[0][1] = 0; lashen1[0][2] = 0; lashen1[0][3] = 0;
        lashen1[1][0] = 0; lashen1[1][1] = 1; lashen1[1][2] = 0; lashen1[1][3] = 0;
        lashen1[2][0] = 3; lashen1[2][1] = 3; lashen1[2][2] = 1; lashen1[2][3] = 0;
        lashen1[3][0] = 0; lashen1[3][1] = 0; lashen1[3][2] = 0; lashen1[3][3] = 1;
        for(var i = 0;i < points.length;i++){          //拉伸前图形点的表示
            var qian3D = Object.create({x:points[i].x,y:points[i].y,z:0});
            points3Dq.push(qian3D);
        }
        for(var i = 0;i < points.length;i++){          //拉伸后图形点的表示
            var hou3D = Object.create({x:points[i].x,y:points[i].y,z:Zvalue});
            points3Dh.push(hou3D);
        }
        for(var i = 0;i < points3Dq.length;i++){       //将二维图形进行拉伸变换
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points3Dq[i].x,points3Dq[i].y,points3Dq[i].z,1];
            var farrpoints = Mul1v4(sarrpoints,lashen1);      //1×4 矩阵
            var newP =
            Object.create({x:farrpoints[0][0],y:farrpoints[0][1],z:farrpoints[0][2]});
            tem1points.push(newP);
        }
        points3Dq.length = 0;
        for(var ii = 0;ii < tem1points.length;ii++) {
            points3Dq.push(tem1points[ii]);
        }
        for(var i = 0;i < points3Dh.length;i++){       //将拉伸后形成的图形的点进行更新
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points3Dh[i].x,points3Dh[i].y,points3Dh[i].z,1];
            var farrpoints = Mul1v4(sarrpoints,lashen1);
            var newP =
            Object.create({x:farrpoints[0][0],y:farrpoints[0][1],z:farrpoints[0][2]});
            tem2points.push(newP);
```

```
            }
            points3Dh.length = 0;
            for(var ii = 0;ii < tem2points.length;ii++) {
                points3Dh.push(tem2points[ii]);
            }
            context.beginPath();                              //将图形的点进行更新变换
            context.moveTo(points3Dq[0].x,points3Dq[0].y);
            for(var l = 1;l < points3Dq.length;l++){
                context.lineTo(points3Dq[l].x,points3Dq[l].y);
            }
            context.moveTo(points3Dh[0].x,points3Dh[0].y);
            for(var l = 1;l < points3Dh.length;l++){
                context.lineTo(points3Dh[l].x,points3Dh[l].y);
            }
            for(var c = 0;c < points3Dq.length;c++) {
                context.moveTo(points3Dq[c].x,points3Dq[c].y);
                context.lineTo(points3Dh[c].x,points3Dh[c].y);
            }
            context.stroke();
        }
    //定义 1×4 矩阵
    function Mul1v4(arr1,arr2){
        var temp2Array = new Array(1);
        temp2Array[0] = new Array(4);
        if(arr1[0].length!= arr2.length){
            alert("数组行列不匹配,无法相乘");
        }else{
        temp2Array[0][0] =
        arr1[0][0] * arr2[0][0] + arr1[0][1] * arr2[1][0] + arr1[0][2] * arr2[2][0] + arr1[0][3] *
    arr2[3][0];
        temp2Array[0][1] =
        arr1[0][0] * arr2[0][1] + arr1[0][1] * arr2[1][1] + arr1[0][2] * arr2[2][1] + arr1[0][3] *
    arr2[3][1];
        temp2Array[0][2] =
        arr1[0][0] * arr2[0][2] + arr1[0][1] * arr2[1][2] + arr1[0][2] * arr2[2][2] + arr1[0][3] *
    arr2[3][2];
        temp2Array[0][3] =
        arr1[0][0] * arr2[0][3] + arr1[0][1] * arr2[1][3] + arr1[0][2] * arr2[2][3] + arr1[0][3] *
    arr2[3][3];
        return temp2Array;
        }
    }
```

添加名为“拉伸”的可以执行拉伸功能的按钮：

```
<input type = "button" name = "lashen" value = "拉伸" onclick = "lashen()" />
```

拾取多边形并单击“拉伸”按钮后，会弹出一个可以输入拉伸距离的对话框，经过设置，默认拉伸距离为30，但是可根据具体情况进行修改。单击“确定”按钮便完成二维图形的拉伸。二维图形经拉伸生成三维图形的效果如图11.2-9所示。



图 11.2-9　二维图形拉伸生成三维图形

需要注意的是，代码中 points3Dq（和 points3Dh）、tem1points（和 tem2points）在这里均表示拉伸前（后）图形的点集。但是 tem1points 为局部变量，只在拉伸变换中使用，表示拉伸前图形的点集；points3Dq 为全局变量，在这里表示拉伸前的图形的点集，但在后面的三维变换中表示三维变换前图形的点集。

11.2.7　三维图形的组合变换

三维图形组合变换和二维图形组合变换的原理基本相同。相对二维图形，三维图形多了纵向坐标，因此，三维坐标在齐次坐标下的变换用 4×4 矩阵表示。

1. 三维图形的平移变换

三维图形的平移变换相关功能代码如下所示：

```
function dMove(){
        var sX;var sY;var fX;var fY;
        var Rnum = 0;
        huabu.onmousedown = function(e){
            if(Rnum == 1){
            e = window.event||e;
            sX = e.pageX - this.offsetLeft;
            sY = e.pageY - this.offsetTop;
            }
            else if(Rnum == 2) {
            e = window.event||e;
            var fX = e.pageX - this.offsetLeft;
            var fY = e.pageY - this.offsetTop;
            var dX = fX - sX;
            var dY = fY - sY;                       //图形在 x、y 轴方向上移动的距离
            for(var a = 0;a < points3Dq.length;a++){  //将拉伸后图形的点进行平移变换
                points3Dq[a].x = points3Dq[a].x + dX;
                points3Dq[a].y = points3Dq[a].y + dY;
            }
            for(var b = 0;b < points3Dh.length;b++){  //将平移后图形的点进行更新变换
                points3Dh[b].x = points3Dh[b].x + dX;
                points3Dh[b].y = points3Dh[b].y + dY;
```

```
            }
            context.clearRect(0,0,mycanvas.width,mycanvas.height);     //更新图形的点
            context.beginPath();
            context.moveTo(points3Dq[0].x,points3Dq[0].y);
            for(var cc = 1;cc < points3Dq.length;cc++){
                context.lineTo(points3Dq[cc].x,points3Dq[cc].y);
            }
            context.moveTo(points3Dh[0].x,points3Dh[0].y);
            for(var cc = 1;cc < points3Dh.length;cc++){
                context.lineTo(points3Dh[cc].x,points3Dh[cc].y);
            }
            for(var c = 0;c < points3Dq.length;c++){
                context.moveTo(points3Dq[c].x,points3Dq[c].y);
                context.lineTo(points3Dh[c].x,points3Dh[c].y);
            }
            context.strokeStyle = "black";
            context.stroke();
        }
    }
```

添加名为“3D 平移”的可以执行三维平移变换相关功能的按钮：

```
<input type = "button" name = "B12" value = "3D 平移" onclick = "dMove()" />
```

进行拉伸形成三维图形后，单击“3D 平移”按钮，便可以完成三维图形的平移变换。平移效果如图 11.2-10 所示。

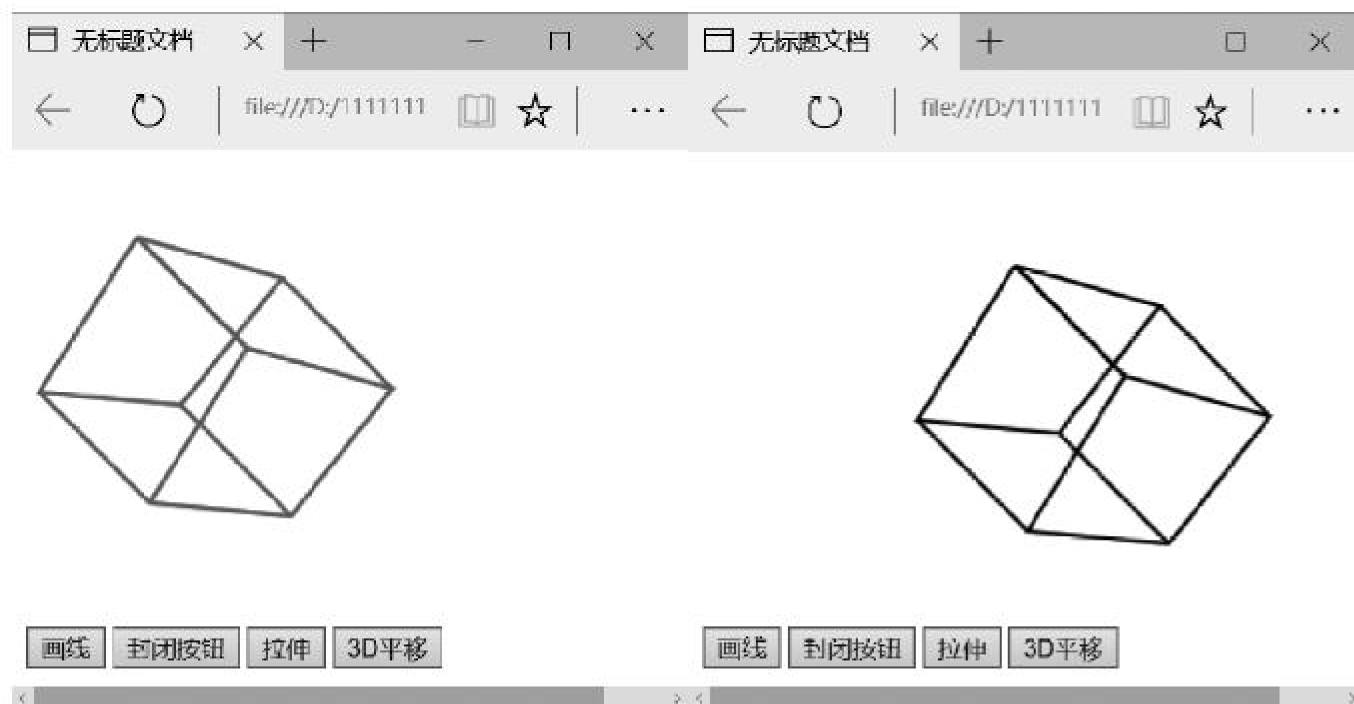


图 11.2-10　三维图形的平移变换

2. 三维图形的旋转变换

在三维空间中，可以分别绕 x、y、z 三个轴进行旋转变换。相关代码如下所示：

```
function dRevolve(){
        var ang  = document.getElementById('xuanzhuanAng').value;  //旋转角度
        var radioVal  =  document.getElementsByName('1');          //旋转轴
        var roller = 0;
        for(var i = 0;i < radioVal.length;i++){
        if(radioVal[i].checked){
        roller  = radioVal[i].value;
```

```
break; }
};
var zheng = Math.sin(ang * Math.PI/180);
var yu = Math.cos(ang * Math.PI/180);
        if(roller == 0){
            var T1 = new Array(4);                    //绕 x 轴旋转变换矩阵
            for(var k = 0;k < T1.length;k++){
                T1[k] = new Array(4);
            }
            T1[0][0] = 1; T1[0][1] = 0; T1[0][2] = 0; T1[0][3] = 0;
            T1[1][0] = 0; T1[1][1] = yu; T1[1][2] = zheng; T1[1][3] = 0;
            T1[2][0] = 0; T1[2][1] = - zheng; T1[2][2] = yu; T1[2][3] = 0;
            T1[3][0] = 0; T1[3][1] = 0; T1[3][2] = 0; T1[3][3] = 1;
        }else if(roller == 1) {
            var T1 = new Array(4);                    //绕 y 轴旋转变换矩阵
            for(var k = 0;k < T1.length;k++){
                T1[k] = new Array(4);
            }
            T1[0][0] = yu; T1[0][1] = 0; [0][2] = - zheng; [0][3] = 0;
            T1[1][0] = 0; T1[1][1] = 1; T1[1][2] = 0; T1[1][3] = 0;
            T1[2][0] = zheng; T1[2][1] = 0; T1[2][2] = yu; T1[2][3] = 0;
            T1[3][0] = 0; T1[3][1] = 0; T1[3][2] = 0; T1[3][3] = 1;
        }else if(roller == 2) {
            var T1 = new Array(4);                    //绕 z 轴旋转变换矩阵
            for(var k = 0;k < T1.length;k++){
                T1[k] = new Array(4);
            }
            T1[0][0] = yu; T1[0][1] = zheng; T1[0][2] = 0; T1[0][3] = 0;
            T1[1][0] = - zheng; T1[1][1] = yu; T1[1][2] = 0; T1[1][3] = 0;
            T1[2][0] = 0; T1[2][1] = 0; T1[2][2] = 1; T1[2][3] = 0;
            T1[3][0] = 0; T1[3][1] = 0; T1[3][2] = 0; T1[3][3] = 1;
        }
        var temP = new Array();
        for(var i = 0;i < points3Dq.length;i++){//三维变换矩阵
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points3Dq[i].x,points3Dq[i].y,points3Dq[i].z,1];
            var farrpoints = Mul1v4(sarrpoints,T1);
            var newP =
        Object.create({x:farrpoints[0][0],y:farrpoints[0][1],z:farrpoints[0][2]});
            temP.push(newP);
        }
        points3Dq.length = 0;
        for(var ii = 0;ii < temP.length;ii++){
            points3Dq.push(temP[ii]);
        }
        var temP1 = new Array();
        for(var i = 0;i < points3Dh.length;i++){//将旋转后图形的点进行更新
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points3Dh[i].x,points3Dh[i].y,points3Dh[i].z,1];
            var farrpoints = Mul1v4(sarrpoints,T1);
            var newP =
```

```
                    Object.create({x:farrpoints[0][0],y:farrpoints[0][1],z:farrpoints[0][2]});
                        temP1.push(newP);
                    }
                    points3Dh.length = 0;
                    for(var ii = 0;ii < temP1.length;ii++) {
                        points3Dh.push(temP1[ii]);
                    }
                    context.clearRect(0,0,mycanvas.width,mycanvas.height);     //更新点集
                    context.beginPath();
                    context.moveTo(points3Dq[0].x,points3Dq[0].y);
                    for(var l = 1;l < points3Dq.length;l++){
                        context.lineTo(points3Dq[l].x,points3Dq[l].y);
                    }
                    context.moveTo(points3Dh[0].x,points3Dh[0].y);
                    for(var l = 1;l < points3Dh.length;l++){
                        context.lineTo(points3Dh[l].x,points3Dh[l].y);
                    }
                    for(var c = 0;c < points3Dq.length;c++){
                        context.moveTo(points3Dq[c].x,points3Dq[c].y);
                        context.lineTo(points3Dh[c].x,points3Dh[c].y);
                    }
                    context.strokeStyle = "blue";
                    context.stroke();
    }
```

添加名为"3D 旋转"的可以执行三维旋转功能的按钮：

```
< input type = "button" name = "dxuanzhaun " value = "3D 旋转" onclick = " dRevolve()" />
旋转角度: < input id = "xuanzhuanAng" type = "text" value = 15 size = "1">
旋转轴: < input id = "Xzhou" type = "radio" checked = "checked" name = "1" value = "0"/> x 轴
< input id = "Yzhou" type = "radio" value = "1" name = "1"/> y 轴
< input id = "Zzhou" type = "radio" value = "2" name = "1"/> z 轴
```

当完成拉伸变换后，单击"3D 旋转"按钮，便完成相关三维图形的旋转变换。经过设置，这里默认旋转角度为 15°，绕 x 轴旋转，但是可根据需要进行更改。绕 x 轴、y 轴的旋转变换效果如图 11.2-11 所示。

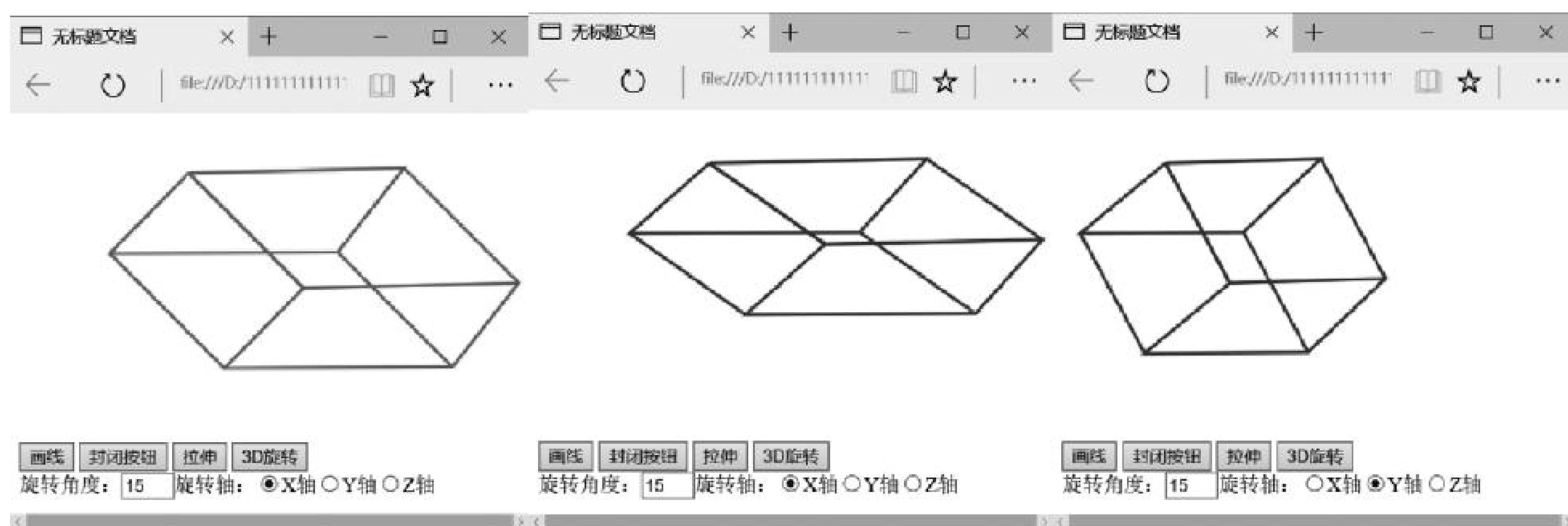


图 11.2-11　三维图形绕 x 轴、y 轴的旋转变换

3. 三维图形的比例变换

三维图形比例变换的变换矩阵具有 x、y、z 三个方向的比例因子。相关功能代码如下所示：

```
function dZoom(){
        var R = window.prompt("请输入比例因子",0.5);
        var r1 = new Array(4);
        for(var i = 0;i < r1.length;i++) {                //比例变换矩阵
            r1[i] = new Array(4);
        }
        r1[0][0] = R; r1[0][1] = 0; r1[0][2] = 0; r1[0][3] = 0;
        r1[1][0] = 0; r1[1][1] = R; r1[1][2] = 0; r1[1][3] = 0;
        r1[2][0] = 0; r1[2][1] = 0; r1[2][2] = R; r1[2][3] = 0;
        r1[3][0] = 0; r1[3][1] = 0; r1[3][2] = 0; r1[3][3] = 1;
        var zoompoints = new Array();
        for(var i = 0;i < points3Dq.length;i++){          //三维变换矩阵
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points3Dq[i].x,points3Dq[i].y,points3Dq[i].z,1];
            var farrpoints = Mul1v4(sarrpoints,r1);
            var newP =
        Object.create({x:farrpoints[0][0],y:farrpoints[0][1],z:farrpoints[0][2]});
            zoompoints.push(newP);
        }
        points3Dq.length = 0;
        for(var j = 0;j < zoompoints.length;j++){
            points3Dq.push(zoompoints[j]);
        }
        var zoompoints1 = new Array();                    //将变换后图形的点进行更新
        for(var i = 0;i < points3Dh.length;i++){
            var sarrpoints = new Array(1);
            sarrpoints[0] = [points3Dh[i].x,points3Dh[i].y,points3Dh[i].z,1];
            var farrpoints = Mul1v4(sarrpoints,r1);
            var newP =
        Object.create({x:farrpoints[0][0],y:farrpoints[0][1],z:farrpoints[0][2]});
            zoompoints1.push(newP);
        }
        points3Dh.length = 0;
        for(var j = 0;j < zoompoints1.length;j++){
            points3Dh.push(zoompoints1[j]);
        }
        context.clearRect(0,0,mycanvas.width,mycanvas.height);    //更新点集
        context.beginPath();
        context.moveTo(points3Dq[0].x,points3Dq[0].y);
        for(var l = 1;l < points3Dq.length;l++){
            context.lineTo(points3Dq[l].x,points3Dq[l].y);
        }
        context.moveTo(points3Dh[0].x,points3Dh[0].y);
        for(var l = 1;l < points3Dh.length;l++){
            context.lineTo(points3Dh[l].x,points3Dh[l].y);
        }
```

```
        for(var c = 0;c < points3Dq.length;c++){
            context.moveTo(points3Dq[c].x,points3Dq[c].y);
            context.lineTo(points3Dh[c].x,points3Dh[c].y);
        }
        context.stroke();
    }
```

添加名为“3D 缩放”的可以执行三维比例变换的按钮：

```
<input type = "button" name = "dsuofang" value = "3D 缩放" onclick = "dZoom()" />
```

完成三维图形后，单击“3D 缩放”按钮便会弹出一个可以输入比例因子的对话框，经设置默认比例因子为 0.5，但是可以根据需要进行更改。三维图形比例变换后的效果如图 11.2-12 所示。

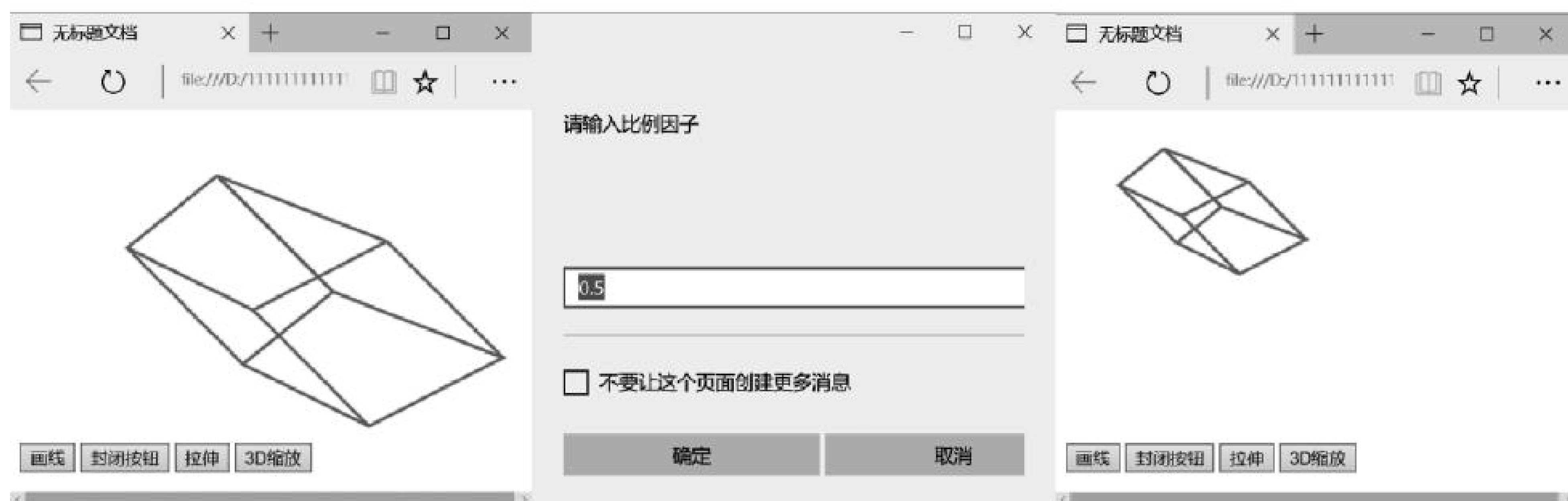


图 11.2-12 三维图形比例变换

由图 11.2-12 可以看出，三维图形的比例变换是以原点为比例变换中心的。这是因为三维图形涉及 z 轴，而纵坐标在这里是无法用鼠标拾取的，所以无法随意拾取图形的比例变换中心，只能默认使用原点。

11.3 基于 WebGL 的 3D 图形

WebGL 是一种 3D 绘图标准，这种绘图技术标准允许把 JavaScript 和 OpenGL ES 2.0 结合在一起，WebGL 可以为 HTML 5 的 canvas 标签提供硬件 3D 加速渲染，这样 Web 开发人员可以借助系统显卡在浏览器中更流畅地展示 3D 场景和模型。

WebGL 完美地解决了现有的 Web 交互式三维动画的两个问题：第一，它通过 HTML 脚本本身实现 Web 交互式三维动画的制作，无须任何浏览器插件支持；第二，它利用底层的图形硬件加速功能进行图形渲染，是通过统一的、标准的、跨平台的 OpenGL 接口实现的。

但是，如果使用原生 WebGL 进行 3D 网页制作，其过程非常烦琐，开发难度很大，会严重影响 WebGL 的开发效率。为了解决这些问题，一些开发人员开发出了许多基于 WebGL 的开源框架，从而为其他开发人员开发类似的程序提供了便利。目前，比较流行的 WebGL 框架有 Three.js、PhiloGL、Babylon.js、SceneJS、WebGLU 等，这些引擎库能够让用户更加

方便地进行3D图形绘制和动画的制作。其中，Three.js是目前应用最广泛的WebGL框架，而且完全采用JavaScript编写而成，以简单、直观的方式封装了3D图形编程中常用的对象，使用了很多图形引擎的高级技巧，极大地提高了性能，它是一款优秀的Web端3D引擎。

11.3.1 Three.js绘制3D图形的结构

Three.js是基于WebGL的3D Javascript库，它封装了场景、相机、几何、3D模型加载器、灯光、材质、着色器、动画、粒子、数学工具等。这样的封装让用户能够更加直观地在网页中制作3D图形和动画。

在Three.js中场景、相机和渲染器是3D图形绘制的基础：场景是所有对象放置和展示的平台；相机决定图形展示的角度；渲染器决定渲染的结果应该画在页面的什么元素上面，并且以怎样的方式来绘制。完成3D图形的基本代码如下所示。

1. 场景的设置

三维场景使用THREE.Scene来设置。

```
varscene = new THREE.Scene();
```

2. 创建相机

Three.js提供了两种相机类型，即透视相机(PersepectiveCamera)和正交相机(OrthographicCamera)。其中透视相机最贴近真实世界，所以使用较为广泛。

```
var camera =
new THREE.PerspectiveCamera(45,window.innerWidth/window.innerHeight,1,1000);
```

3. 创建渲染器

为了达到更好的展示效果，一般选用Three.js中渲染效果较好的WebGLRenderer，其兼容性不是最优，但是也可以满足要求。

```
var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth,window.innerHeight);
```

4. 设置光源

在Three.js中提供了4种基本光源，即环境光(AmbientLight)、点光源(PointLight)、聚光灯(SpotLight)和方向光(DirectionLight)。系统中主要用到环境光、点光源和聚光灯。

```
var ambientLight =  THREE.AmbientLight();
```

完成以上四步后，一个三维网页的骨架就搭建成功了，之后便可以在其中绘制构建三维网格模型、添加材质以及灯光等效果了。

5. 绘制图形

三维模型是由三角形面或者四边形面组成的网格模型。在Three.js中用THREE.Mesh来表示三维网格模型。THREE.Mesh的构造函数是：THREE.Mesh = function(geometry,material)。其中第一个参数geometry是一个THREE.Geometry类型的对象，包含了模型顶点之间的连接关系；第二个参数material定义了模型的材质，材质会影响光

照、纹理对 Mesh 的作用效果。

```
var mesh = new THREE.Mesh(geometry,material);
```

6. 渲染循环

使创建的三维模型可以在场景中渲染循环,形成动画效果。

```
requestAnimationFrame(render);
renderer.render(scene,camera);
```

11.3.2 Web 下的三维模型的显示

根据上述 Three.js 绘制 3D 图形的基本结构,现通过一个简单例子来真正实现网页中的三维模型的显示。相关代码如下所示:

```
<script src = "three.js"></script>
<script>
//场景
var scene = new THREE.Scene();
//相机
var camera =
new THREE.PerspectiveCamera(45,window.innerWidth/window.innerHeight,1,1000);
camera.position.x = 15;                              //设置相机的位置
camera.position.z = 15;
//渲染
var renderer = new THREE.WebGLRenderer();
renderer.setClearColor("#FFFFFF");                   //渲染器背景,若不设置,背景默认为黑色
renderer.setSize(window.innerWidth,window.innerHeight);   //渲染器大小,为计算机屏幕大小
document.body.appendChild(renderer.domElement);      //渲染器结果,将结果加载到界面中
//几何体
var geometry = new THREE.CubeGeometry(2,2,2);        //设置对象
var material = new THREE.MeshBasicMaterial({color:0x44ff44});   //设置立方体的材质颜色
var cube = new THREE.Mesh(geometry,material);        //将对象和材质传递给几何体
scene.add(cube);                                     //将几何体加载到场景中
//光源
var ambientLight = new THREE.AmbientLight(0x0c0c0c); //环境光
scene.add(ambientLight);
function render(){
    requestAnimationFrame(render);                   //渲染循环
    cube.rotation.x += 0.1;                          //立方体旋转速度
    cube.rotation.y += 0.1;
    cube.rotation.z += 0.1;
    renderer.render(scene,camera);                   //使用渲染器,结合相机和场景得到结果画面
}
render();
</script>
```

至此一个简单的立方体便完成了,其在网页中的显示效果如图 11.3-1 所示。

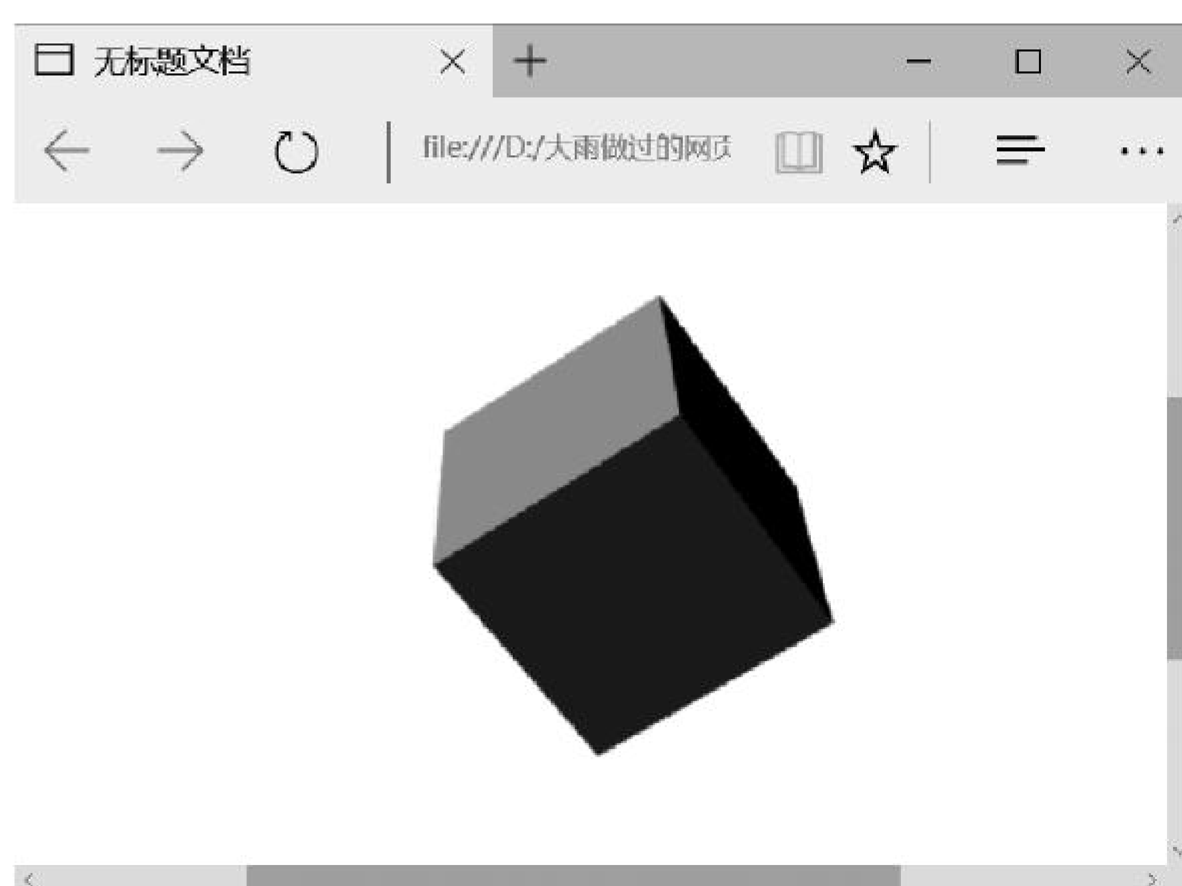


图 11.3-1 网页中的立方体

需要注意的是，WebGL 对浏览器的要求特别高，目前支持 WebGL 的浏览器有 Chrome、FireFox、360 安全浏览器 6.0、IE9 等。

参 考 文 献

[1] 何援军. 计算机图形学[M]. 2 版. 北京：机械工业出版社，2016.
[2] 和青芳. 计算机图形学原理及算法教程(Visual C++版)[M]. 北京：清华大学出版社，2010.
[3] 孔令德. 计算机图形学基础教程(VC++版)[M]. 北京：清华大学出版社，2009.
[4] 孙家广，胡事民. 计算机图形学基础教程[M]. 北京：清华大学出版社，2005.
[5] 和克智. OpenGL 编程技术详解[M]. 北京：化学工业出版社，2010.
[6] 帕里西. WebGL 入门指南[M]. 北京：人民邮电出版社，2013.
[7] 孙家广. 计算机图形学[M]. 北京：清华大学出版社，1998.
[8] 潘云鹤. 计算机图形学原理、方法及应用[M]. 北京：高等教育出版社，2003.
[9] SAHNI S. 数据结构、算法与应用——C++语言描述[M]. 汪诗林，孙晓东，译. 北京：机械工业出版社，2000.
[10] SHRENER D，等. OpenGL 编程指南[M]. 李军，徐波，译. 北京：机械工业出版社，2010.
[11] WILTON J P. Javascript 入门经典[M]. 胡献慧，译. 北京：清华大学出版社，2017.
[12] WRIGHT R S，LIPCHAK B，HAEMEL N. OpenGL 超级宝典[M]. 5 版. 付飞，李艳辉，译. 北京：人民邮电出版社，2012.
[13] 唐荣锡，汪嘉业，彭群生. 计算机图形学教程[M]. 北京：科学出版社，2003.
[14] 于海燕，蔡鸿明，何援军. 图学计算基础[J]. 图学学报，2015(3)：1-10.
[15] 陆玲. 计算机图形学[M]. 北京：机械工业出版社，2017.
[16] 于万波. 计算机图形学：VC++实现[M]. 2 版. 北京：清华大学出版社，2017.
[17] 赵辉，王晓玲. 计算机图形学：OpenGL 三维渲染：C# 版[M]. 北京：清华大学出版社，2016.
[18] 黄静. 计算机图形学及其实践教程[M]. 北京：机械工业出版社，2015.
[19] 王志喜，何勇. 计算机图形学[M]. 北京：中国矿业大学出版社，2013.
[20] HEARN D，BAKER M P，CARITHERS W R. 计算机图形学[M]. 蔡士杰，杨若瑜，译. 北京：电子工业出版社，2014.
[21] 王汝传. 计算机图形学教程[M]. 3 版. 北京：人民邮电出版社，2014.
[22] 肖嵩，杜建超. 计算机图形学原理及应用[M]. 西安：西安电子科技大学出版社，2014.
[23] 徐文鹏. 计算机图形学基础：OpenGL 版[M]. 北京：清华大学出版社，2014.
[24] 杜晓增，丁宇辰. 计算机图形学基础[M]. 2 版. 北京：机械工业出版社，2013.
[25] 贾建，康宝生. 计算机图形学[M]. 西安：西安电子科技大学出版社，2013.
[26] 魏海涛. 计算机图形学：理论、工具与应用[M]. 3 版. 北京：电子工业出版社，2013.
[27] 孔令德. 计算机图形学实践教程：Visual C++版[M]. 2 版. 北京：清华大学出版社，2013.
[28] 郭际元，曾文，龚君芳. 计算机图形学实习教程[M]. 北京：中国地质大学出版社，2012.